University of Zurich[UZH]

Department of Informatics

Efficient Stream-Processing of Large Geometric Data Sets

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Sciences

by
Jonas Boesch
Master of Science in Computer Science, UZH

Accepted on the recommendation of

Prof. Dr. R. Pajarola
Prof. Dr. Matthias Zwicker
Prof. Dr. Yanci Zhang
Prof. Dr. Harald Gall

2011

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 27, 2010

The head of the Ph.D. program in informatics: Prof. Abraham Bernstein, Ph.D.

ABSTRACT

Points as rendering and modeling primitives have become a powerful alternative to polygonal object representation. Three-dimensional point samples are the fundamental geometry-defining entities, and are the natural raw output data primitives of 3D scanning and reconstruction systems. With the continually increasing density and extent of raw point cloud data, effective algorithms and systems are required to cope efficiently with the massive amounts of point samples.

This thesis presents a system for streaming large geometric data sets. New methods to compute geometric properties such as Gaussian curvature are presented and analyzed. Combined with various existing algorithms, they form a complete geometry processing solution, including a method to compute and embed discrete reeb graphs for point sets. The integrated algorithms are discussed with respect to quality and performance.

Novel concepts and techniques used for efficient geometric processing such as stream operators and run-time structures are presented. Stream operators encapsulate algorithms that operate on the geometry of points or polygonal meshes, read and write arbitrary attributes and can be chained in any order where the input requirements are met. Run-time structures enable efficient and type-safe access to the attributes of the streamed data.

Finally, a streaming mesh simplification system based on the $\epsilon - net$ sampling method is presented. We show that our method is capable of efficiently processing large polygonal meshes, decrease them in size while minimizing simplification artifacts.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to Prof. Dr. Renato Pajarola, for being a great supervisor and for his support and guidance throughout my thesis, in technical matters and beyond.

I would like to extend my appreciation to the members of my thesis committee: Prof. Dr. Matthias Zwicker and Prof. Dr. Yanci Zhang.

Many thanks go to the previous and current members of the VMML lab, especially to Stefan Eilemann, Thomas Hübner and Susanne Suter, for our discussions on technical matters, and everything else.

I would also like to express my deepest gratitude to my family: my brother Michael Elias, my sisters Rahel and Vera Maria and my parents Jakob and Corinne. Thank you for always being there for me. I would not have managed this without you. I would like to extend my appreciation to Gert, to my new brother Slava, to his mother Nina, and, last but not least, to my girlfriend Helene.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

C H A P T E R 1

INTRODUCTION

1.1 Overview

Points as rendering and modeling primitives have become a powerful alternative to polygonal object representation [Pfister and Gross, 2004; Gross, 2006; Gross and Pfister, 2007b]. Three-dimensional point samples are the fundamental geometry-defining entities, and are the natural raw output data primitives of 3D scanning and reconstruction systems. Satisfying provably correct sampling criteria as discussed in [Meenakshisundaram, 2001], a set of 3D points fully defines the geometry as well as the topology of a surface including boundaries, components and genus.

With the continually increasing density and extent of raw point cloud data, effective algorithms and systems are required to cope efficiently with the massive amounts of point samples. These operations can only be performed efficiently on large data if memory trashing [Denning, 1970] is avoided. Therefore, data must be paged efficiently into main memory and processed coherently with respect to randomly accessing memory locations.

In [Pajarola, 2005] the concept of stream-processing point data was introduced. The basic idea is to sequentialize the unorganized raw input point data and then feed the resulting point stream through a pipeline of local stream operators. A stream operator encapsulates an algorithm that works on a point and its local neighborhood. This thesis builds on the basic idea of stream processing and extends and improves it with important additions and new ideas. The result of that effort was the creation of the stream_process framework.

1.2 Challenges

Stream Processing of Points Modern laser scanners can produce data sets of millions of points. In order to use this data, it has to be processed, as the output usually consists of only point positions and potentially color information. As the size of these point data sets may exceed the main memory of a modern computer system, ways to process it have been devised. The work of [Pajarola, 2005] introduces the concept of stream-processing point data by sequentializing the raw point clouds in an external preprocess, then loading it point by point into main memory and compute the required point attributes by running it into a series of interdependent operator modules.

Some of the problems of the approach presented in [Pajarola, 2005] are the compile-time dependency of the chosen operator chain, the impossibility to parallelize the processing and the non-inclusiveness of the preprocess.

With a growing number of available operators, the compile-time dependency becomes prohibitive, as a separate executable would have to be produced for all permutations of the processing chain. However, removing this compile-time de-

pendency opens up new problems. A way for each operator to access its required attributes has to be devised that is efficient, type-safe as well as simple to use for the developer.

Modern computers generally have multiple processors with multiple cores. For efficient processing of large data sets, multi-threading should be used to take advantage of the parallelization capabilities. However, multi-threading introduces a great number of difficult issues with respect to synchronization between reading and writing data. Not only do synchronization issues threaten to use up the performance benefits of multi-threading, data integrity might be violated if synchronization is not strictly enforced by the system. Such violations often appear in the form of race conditions which are hard to debug as they do not occur consistently but depend on external circumstances such as processor load. Ideally, a stream-processing system would hide this complexity from the operator programmer.

C++ is a statically typed language. As such, the assigned memory for each class is defined at compile-time. For a configurable operator chain, where any number of operators with corresponding point attributes might be used, an efficient way to access this memory has to be devised. While the naive solution of using void pointers, raw memory blocks and casting might seem appealing to some, this approach lacks type safety and is hard and unintuitive to use for operator developers.

Operators and Algorithms On the algorithmic side, there is a need for new or improved algorithms to compute the geometric properties of points. Many different algorithms to compute attributes such as the normal vector [Alexa et al., 2001; Pauly et al., 2002a; Pauly et al., 2003; Mitra et al., 2004; Alexa and Adamson, 2009] and the principal curvatures and principal curvature directions [Rusinkiewicz, 2004; Pajarola, 2005; Agam and Tang, 2005], are known, each with various benefits and disadvantages. A useful point processing system should therefore allow a user to choose the appropriate algorithm for the situation at hand.

Global Data Structures with Local Data The computation of some properties does require global information and cannot be computed with purely local information, such as data set spanning graphs. Using such algorithms on models whose size exceeds main memory opens up a new set of problems.

One such graph is the reeb graph [Reeb, 1946]. A reeb graph describes the connectivity of level sets or contours for a smooth function defined on a manifold, and so represents an abstraction of the topology. This is useful for a wide variety of applications such as finding topologically similar geometric models [Hilaga et al., 2001; Funkhouser et al., 2005; Steiner and Fischer, 2001], for topological

simplification [Guskov and Wood, 2001; Wood et al., 2004], for computer aided geometric design [Shinagawa et al., 1995] and level set computation [Carr et al., 2004]. Other uses are finding transfer functions for volume rendering [Bajaj et al., 1997; Weber et al., 2007] and operations on surfaces such as compression, reconstruction, embedding and parametrization [Shinagawa et al., 1991; Takahashi et al., 1997; Biasotti et al., 2000; Attene et al., 2001; Biasotti and Ricerche, 2001; Hétroy and Attali, 2003; Zhang et al., 2005]. While approaches have been presented that use streaming in combination with the graph computation [Pascucci et al., 2007], these use triangles or other polygonal meshes and do not operate directly on points.

Polygonal Mesh Sampling Polygon mesh sampling is important in many geometry processing problems, including shape approximation [Heckbert and Garland, 1997; Luebke, 2001], surface reconstruction [Dey, 2006; Amenta et al., 1998; Dey and Goswami, 2004; Gopi et al., 2000; Bernardini et al., 1999] and parameterization [Floater, 1997; Praun et al., 2001; Sorkine et al., 2002; Khodakovsky et al., 2003; Sander et al., 2003; YBS, 2004]. With growing model sizes, streaming approaches are increasingly necessary in order to run sampling algorithms. As a polygonal mesh can be seen as a set of additional constraints for a point set, an extension of the stream processing system to include support for faces would allow for using the stream processing technology for mesh sampling and simplification problems.

1.3 Contributions

The work presented in this thesis solves many of the issues presented in Section 1.2. After presenting the fundamental ideas behind stream processing, we discuss the approaches taken to improve on the seminal work of [Pajarola, 2005] and present a series of novel extensions and enhancements.

A major contribution of this work is the development of the dynamic, fully configurable and extensible stream processing system. As a class framework, it can be utilized as a library or directly by the user through a command-line interface [Bösch and Pajarola, 2008] or a web page [Van Loon, 2007].

In order for the stream processing system to allow this level of configurability, the concept of run-time structs or dynamic classes has been developed, comprising stream_data, attributes, attribute accessors and stream structures. By using C++ templates and a set of base classes for stream operators, point attributes can be created and accessed within an operator without sacrificing either type safety or efficiency. And as the whole processing pipeline is templatized, processing can be performed in single precision, in double pre-

cision or in a mixed-mode where the pipeline generally uses single-precision but precision-critical computations are done in double precision. A novel concept, the chain_operator, has been developed and implemented. While a stream_operator is only aware of a subset of all points in memory, the new chain operators spans the whole active set and can access any point currently loaded into memory.

Efficiency while processing large data sets is crucial. As most modern computers have multiple processors with multiple cores each, multi-threading enables the use of this processing power. In the stream processing system, multi-threading is fully integrated, in a fashion that is transparent to operators. This means that with an operator, a developer does not have to worry about synchronization, locks and other difficulties that generally accompany parallelized processing. A system of reference limits has been developed and integrated in the stream processing system to guarantee that all synchronization issues can be handled transparently.

The computation of geometric properties of points is the main goal of the stream processor. Algorithms in the system have been improved, and alternatives that enhance some properties of the original algorithms have been integrated. The neighborhood search based on spatial trees offers new, more efficient tree types, and the exact type of tree and various tree parameters can be specified at runtime. In addition, it is possible to add alternative tree types without having to change the neighborhood search algorithm. Normal estimation has been enhanced by integrating the natural normals approach proposed by [Alexa and Adamson, 2009], and various approaches to improve curvature estimation have been examined and implemented.

A reeb graph contains information about the contours or connectivity of a model. An stream_operator that computes a reeb graph of a point set model has been integrated into the stream processing system. Algorithms to compute an embedding of the reeb graph into the model have been developed, with different strategies to compute the actual embedded vertex positions. The reeb graph computation can be performed using either a stream_operator or a chain_operator.

While three-dimensional points are the fundamental geometric entities, many algorithms operate on polygonal meshes, and the processing of large meshes poses similar problems as the processing of large point sets. Many of the concepts and technologies developed for point streaming have been used for a streaming mesh simplification tool. The novel approaches required for polygonal mesh streaming were back-ported into the stream processing system. A design that allows multiple simultaneous streams with interdependencies to be processed has been developed, and stream operators have been extended to access data elements of any of the streams in the system.

1.4 Thesis Structure

Chapter 2 details the stream processing system and explains the novel concepts developed for efficient streaming and processing of large point data sets. The design and the abstractions and some of the optimizations implemented to improve the performance of the stream processing system are presented.

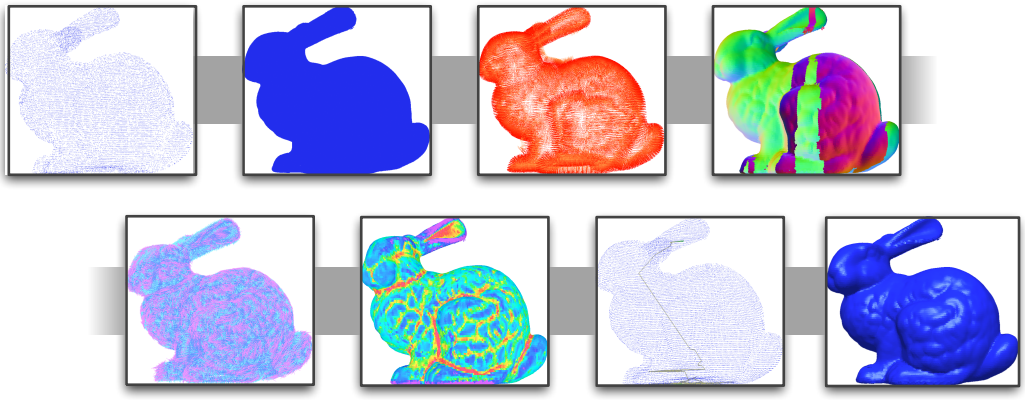
In Chapter 3, the mathematical and geometric theory and the algorithms used in the different operators implemented in the stream processing system are described. Each operator is presented with a short background, implementation details, and the results obtained by using it.

The concepts and technologies developed for the stream processing system have been extended to be useable with polygonal mesh models. Chapter 4 presents a system to simplify polygonal meshes that has been extended to process large models into a streaming mesh simplification library.

Chapter 5 offers a summary and discussion of the results of this thesis, concluding remarks and a view into possible future additions, extensions and improvements.

CHAPTER 2

THE STREAM PROCESSING FRAMEWORK



2.1 Overview

This chapter presents an overview of stream processing and the stream operator concept. The design and novel ideas behind the stream processing system are explained, with a focus on the run-time structures concept proposed in [Bösch and Pajarola, 2009; Bösch and Pajarola, 2008]. Methods required for efficiently processing large amounts of point data are discussed from a technical perspective, and the different parts of the system are presented. The chapter concludes with results on the run-time performance, memory usage and multi-threading behavior of the system.

2.2 Background

2.2.1 Sequential Processing

The basic idea behind stream-processing point data is to order and process the data sequentially in such a way that

i) points can be read from an input-stream into main memory in packets

ii) sets of so called *active* points in main memory can efficiently be processed independently [1]

iii) points are written to an output-stream as soon as they have been fully processed and are no longer required by dependent points

Figure 2.1 illustrates this basic concept of a sliding-window over the set of input points $\mathbf{p}_1, \ldots, \mathbf{p}_n \in \mathbb{R}^3$ as introduced in [Pajarola, 2005]. Since all data processing is limited to the points in the active working set $\mathcal{A}$, at any time only a very limited fraction of data is kept in main memory, which together with the sequential processing supports efficient out-of-core operation on huge point data sets.

The set $\mathcal{A}$ is basically a FIFO queue keeping only the m currently active points $\mathcal{A} = \mathbf{p}_{j-m}, \ldots, \mathbf{p}_m$ in main memory which are to be processed by a chain of local stream operators. As soon as $\mathbf{p}_{j-m} \in \mathcal{A}$ has been processed and is not required by an operation on any subsequent point $\mathbf{p}_{i>j-m}$ it can safely be output.

Since raw point data sets rarely come in a spatially ordered sequence, a sorting process is required to linearly order them. Given an ordering measure along one direction in space, such sorting can efficiently be achieved for very large data by external sort techniques [Linderman, 1996; Knuth, 1998; Vitter, 2001]. Our solution is presented in Section 2.4.1.

[1] the dependency is strictly limited to a well defined local spatial neighborhood relation

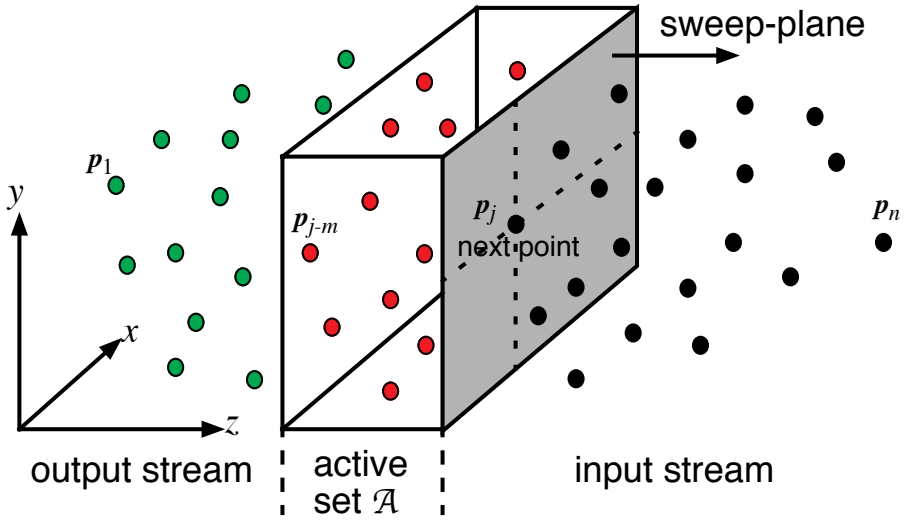


Figure 2.1: *Window of an active set of points sliding over the stream of input point data.*

2.2.2 Stream Operators

The operations supported in the above described stream-processing framework are defined as local operators $\Phi(\mathbf{p}_i)$ that perform a computation on a point $\mathbf{p}_i$ and its attributes only taking the point $\mathbf{p}_i$ itself and a limited set of neighbors $\mathbf{p}_j$ into account. The neighborhood $\mathcal{N}_i$ is typically defined as a k-nearest neighbors or points $\mathbf{p}_j$ within a given range r. The attributes A_i associated with a point $\mathbf{p}_i$ can include a wide range of parameters such as color, normal orientation or curvature. From this definition it is clear that a local operator $\Phi(\mathbf{p}_i)$ can be applied to any point if $\mathbf{p}_i$ itself as well as all neighbors $\mathcal{N}_i$ are part of the current working set $\mathcal{A}$.

Encapsulation

An operator can so encapsulate local operations on the geometry of points. Each stream operator Φ_x is independent of any other operator $\Phi_1, \ldots, \Phi_p$ except for its input requirements. As an example, an operator Φ_{normal} to compute a surface normal of a point might require the vertex position of the current point and the positions of its neighbors. As long as these inputs requirements are fulfilled, there are no other dependencies.

2.2.3 Processing Chain

The stream-processing framework is designed to chain together a series of stream operators $\Phi_1, \ldots, \Phi_p$ that are applied in succession to a stream of points as illustrated in Figure 2.2. Each stream operator Φ_k itself acts as a FIFO queue, passing the points from one to the next operator.

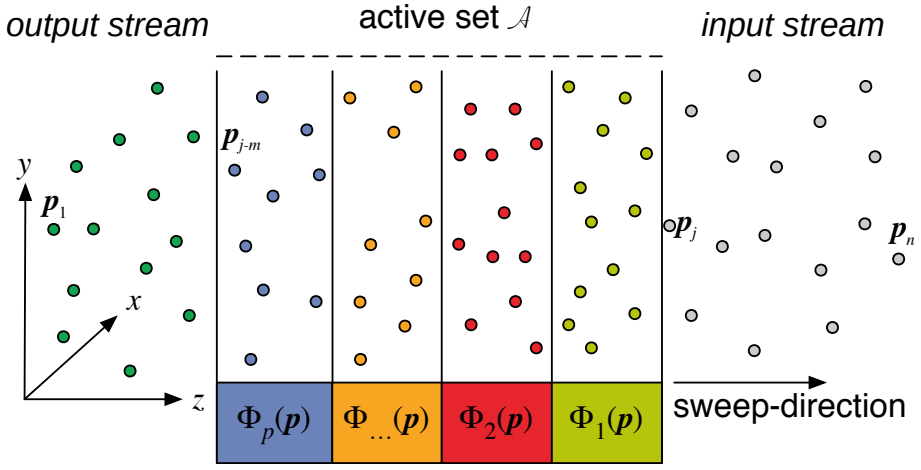


Figure 2.2: *Chain of streamable operators acting on the points passing through the active set.*

Runtime dependency

This concept postulates that a stream operator $\Phi_k(\mathbf{p}_i)$ can be executed on $\mathbf{p}_i$ as soon as no preceding operator $\Phi_{l<k}$ modifies any neighbor points $\mathbf{p}_j \in \mathcal{N}_i$ anymore, or still depends on $\mathbf{p}_i$ for its completion. Moreover, each stream operator Φ_k only passes a point $\mathbf{p}_i$ to the next operator Φ_{k+1} if the point and its attributes have fully been processed. For performance reasons, points can be grouped into packets, and all dependency checks during runtime can be performed only on the packets.

2.2.4 Chain Operators

A chain operator Ψ in scope overarches the entire chain of individual local stream operators Φ. They are intended for active-set global operations that require knowledge of all the elements in the chain of stream operators $\Phi_1, \ldots, \Phi_p$ as illustrated in Figure 2.3. The chain_operator extension to the operator concept was first proposed in [Bösch and Pajarola, 2009].

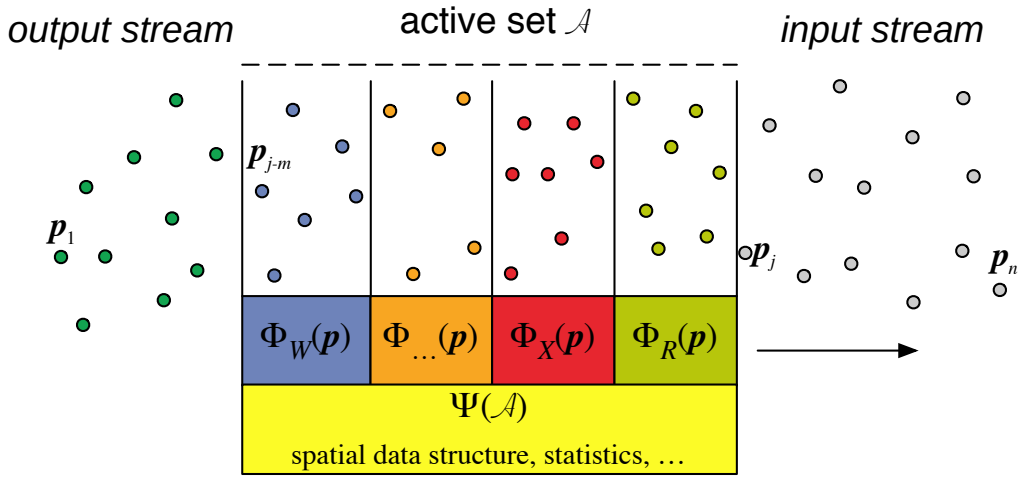


Figure 2.3: *Conceptual diagram stream operators Φ_k, chain operators Ψ and the input set, active set and output set of streamed points.*

2.3 System Architecture

2.3.1 Operator Types

Based on the operator concept as described in Section 2.2.2, this section presents an overview of the actual implementation used in the stream processing system.

Base Operator

The operator_base class is provided as a parent class for all stream_process operators that use stream_data functionality. All stream operators and chain operators must inherit from operator_base. It provides the functionality to easily specify and reserve attributes and collects dependencies on fields of previous operators. Important parts of the C++ interface are shown in Figure 2.4, and consist of all the methods required to setup an stream_data element and its attributes as detailed in section 2.3.2.10. The class is adapted and specialized version of the rt_struct_user class described in [Bösch and Pajarola, 2009].

Stream Operator

A stream operator Φ encapsulates local operations on the geometry of a point p_i and its attributes. Depending on the type of algorithm in the stream_operator, it might access only the current point p_i, or the current point p_i and its neighbors p_{j0},

```
// if an operator needs to create another op
virtual void prepare_setup() {}

// negotiate stream and op config
virtual void setup_negotiate()  {}
// setup according to negotiated settings
virtual void setup_finalize()   {}

// reserve the attributes created in this algorithm
virtual void setup_attributes() {}
virtual void finalize_attributes() {}
// setup the accessors with the final point structure
virtual void setup_accessors()  {}

// shutdown functions - will be called in this order
// - clear stage
// ... finish processing of all points
// - prepare shutdown

virtual void clear_stage()      {}
virtual void prepare_shutdown() {}
```

Figure 2.4: *Abridged C++ interface of the operator base class.*

p_{j1}, ... p_{jn}, or the current point p_i, its neighbors p_{j0}, p_{j1}, ... p_{jn} and it's neighbors' neighbors p_{j00}, p_{j01}, ... p_{jn0}, p_{jn1}, ... p_{jnn}. These access types correspond to the reference limits detailed in Section 2.3.2.8.

It is implemented as a child class of the operator_base class, and uses an interface as shown in Figure 2.5. The actual reference limit checking is done transparently by the stream processing system, the operator only has to set a flag for its access type.

```
virtual void            push( slice_type* data_slice_ );
virtual slice_type*     top();
virtual void            pop();

virtual bool needs_bounds_checking() const { return true; };
```

Figure 2.5: *Abridged C++ interface of the stream operator class.*

Chain Operator

A chain operator Ψ is aware of the complete active set of points in memory. It is notified when a slice of new points is inserted into the active set and when a slice of points has finished processing and will be written out. Any write operations on the points outside of these two events has to be done in an associated stream_operator. Performance statistics are implemented as a chain operator, and a variant of the neighborhood detection is implemented as a pair of chain_operator Ψ and a stream_operator Φ.

```
virtual void insert( slice_type* data_slice_ )  = 0;
virtual void remove( slice_type* data_slice_ )  = 0;
```

Figure 2.6: *Abridged C++ interface of the operator base class.*

2.3.2 Run-time Configurability

In previous stream-processing approaches such as [Pajarola, 2005], the operator chain was set up at compile-time. A stream operator defined a set of per-point attribute parameters that it depends on or modifies, some auxiliary data fields used while executing the operator and some attributes that it adds permanently to a point element, which are added to the output stream at the end of the stream operator chain. Every stream operator defines a struct that contains members variables for all required data, and the temporary as well as the final stream-point structures are defined by multiple inheritance. While the main advantage of this approach is its simplicity, it also lacks in flexibility and consistency. Separate executables for all possible combinatorial configurations of the different stream operators are necessary, which theoretically grows exponentially by 2^p with the number p of operators. For an increasing and extensible library of stream operators this is clearly a limiting constraint. Furthermore, there is no automatic mechanism to verify consistency of attribute fields, e.g. such as ensuring that an attribute xy which operator X depends on is provided by another operator Y. A similar problem arises with the order of operators: While including the normal attribute field allows the use of a curvature operator at compile time, it does not make sure that the normal estimation operator is actually applied first in the chain of stream operators.

Therefore, our stream-processing framework was designed to allow setting up an arbitrary operator chain at run-time, by using either an external configuration file or by specifying stream operators as command line arguments. Also, since attribute fields of different operators are now specified dynamically at run-time, a

registration mechanism can verify that no required attributes are missing, and that the operators are specified in a compatible order.

In this framework, dependencies between stream operators are defined solely by dependencies on certain data elements. This has the advantage that every operator can be replaced as long as the replacement operator can generate the same output data fields. Additionally, since there are no direct code-level dependencies between operators, new operators can be integrated by loading them as separate plugins or dynamic shared libraries at run-time.

2.3.2.1 Run-time Configurable Objects in C++

One approach for defining run-time-configurable objects and attribute fields is to use a container, e.g. a std::vector<>, of boost::any* or similar any-type-objects [Henney, 2000]. In that case, the data members of a point $\mathbf{p}$ are indicated by an index into that vector of boost::any elements, and the element is accessed by casting it to the correct type before use.

The above mentioned solution is quite simple, however, has some important drawbacks. First, the type of the i-th variable field $\mathbf{p}[i]$ is not defined in the header of the operator. boost::any instances are generic, and the type is defined by its first assignment. This means that an external specification for the type of each attribute field is required. Furthermore, in the case of accessing an attribute field using a wrong type, the user will not receive a compile-time error or warning. Instead, an exception will be thrown at run-time. In the context of stream-processing, each attribute field has a fixed type. Therefore, the boost::any approach would not be well suited, in contrast to the dynamic structures that we describe below, where attribute fields can be declared in the stream operator header in a way similar to a normal C++ variable. Moreover, boost::any objects are not compatible with memory pooling. Only the any-class wrapper object can be allocated beforehand, but not the actual data, since it is generated on first assignment. This is because the exact type is only specified at run-time, on first access, and it is therefore impossible to determine at compile-time. Finally, boost::any uses *run-time type information* (RTTI) to determine the type on every access, which might introduce an additional performance penalty. In a naive implementation, each access might even require a string comparison.

Another alternative would be to just use void* pointers, custom allocated memory fields, type lookup tables, and constant casting by the programmer. This system is easy to implement, but hard to use and quite error prone in practice, as there is neither any kind of type safety, nor a reliable way to detect wrongly assigned types during either compile- or run-time.

2.3.2.2 Run-time Structures with stream_data

We designed a new run-time structures concept for streaming data that enables the use of type-safe and run-time configurable yet efficient structures with member variables.

The framework is designed as a collection of classes and class templates. We will start with an overview of the system, and then discuss the most important parts in more detail.

A stream is a collection of stream_data objects that comprise a model. The primary stream is always points or vertices, but secondary or tertiary streams might contain information on faces such as triangles or quads or other data. Each stream is defined by its stream_structure, which is an ordered collection of the different attribute objects associated with each stream_data object. For a point, these attributes might be vertex attributes such as position, normal and color.

A stream_operator or chain_operator encapsulates an algorithm to be run on the stream_data. In order to access the data stored in the stream_data object, attribute accessors are used. attribute_accessor is a template and is parametrized with the desired object in the stream_operator. As attribute_accessor is a functor, performing read and write operations on to a stream_data from within an stream_operator is simple and intuitive.

Finally, a slice is a container object that holds a spatial slice of the whole model. It has a container of stream_data objects for each stream and additional boundary information. The active set is the set of slices currently loaded into memory.

All classes are defined in the stream_process namespace to avoid name collisions.

A more in-depth explanation of the different classes and their responsibilities is presented as follows

2.3.2.3 stream_data

The stream_data structure is defined as a class with no attribute fields. However, at run-time, each stream_data instance is allocated enough memory to hold all member attributes. Basically, it represents the storage of a point while it is in main memory as part of the active set. The standard and copy-constructor and assignment operators are kept private to force the use of the factory class for instantiating stream_data. The factory classes can query the stream_structure instance for each stream and, therefore, know the exact size of the stream_data type of each stream.

2.3.2.4 attribute

An attribute object contains meta data about a single attribute field or data member of a stream_data, which includes its name, type, and various flags. One flag determines the input-output status of an attribute, that is, if its data that was read from the source stream, and if the data is just used during processing or if it should be written to the output stream. Another flag determines the number of objects. This makes it possible for an attribute to be an array, that is, it can store more than a single element of the same type. This simplifies the use of arrays, as otherwise a user would have to define an attribute for every single element in the array.

2.3.2.5 stream_structure

The stream_structure class contains information on all the dynamic attribute data fields in the form of attribute objects. It is used by stream- or chain-operator-based classes to register and query the set of member fields that are required for its encapsulated algorithm, and by the factory to compute the memory usage of the respective stream_data. Each stream contains a single stream_structure object. Member fields can only be registered in the setup stage, as soon as processing begins, the stream_structure is fixed and cannot be changed anymore. More information on this can be found in section 2.3.2.10.

2.3.2.6 attribute_accessor

An attribute_accessor object is used to access an attribute field or member variable of a stream point during the processing stage. It is a template object, with the template parameter being the type of the variable that the attribute_accessor enables access to. Each attribute accessor contains a single member variable, the offset to the start of its data object within the memory associated with the stream_data. This offset is set automatically be the factory after the type of all member fields within each stream's stream_data object have been validated. Note that the compiler will issue a warning when automatic casting cannot be done safely (e.g. assigning a signed to an unsigned integer) or will report a compilation error when automatic casting cannot be done (e.g. when trying to assign a reference of the wrong type). This behavior is consistent with a programmers' experience, and is the same as when using normal variables.

2.3.2.7 slice

A slice is a container of stream_data pointers and some additional information such as minimum and maximum references of stream_data elements to other stream_data elements. By using slices, we can group stream_data instances

together, which allows for efficient multi-threading as well as reducing the number of dependency checks. The references that are stored within a slice are ranges in which other points might be accessed during the processing of this slice. Using the default slice type, two kinds of ranges are computed: the direct neighbor reference limits, and the ring one reference limits.

2.3.2.8 reference limits

single point reference limits Operators that do not require any neighborhood information do not have to track any reference limits, and respectively, the minimal and maximal reference limits are the current point p_i's coordinate on the streaming axis.

neighbor reference limits Given a set of points consisting of p_i and its neighboring points p_{j0}, p_{j1}, ... p_{jn}, the direct neighbor reference limits are the minimal and maximal coordinate on the streaming axis of any point in the set. Any operator that only works on a point an its neighbors can safely operate even in a multithreading environment as long as no other thread is active within the neighbor reference limits.

ring one reference limits Given a set of points consisting of p_i its neighboring points p_{j0}, p_{j1}, ... p_{jn} and the neighbors of each neighboring point p_{j0}, p_{j1}, ... p_{jn0}, p_{jn1}, ... p_{jnn}, the ring one reference limits are the minimal and maximal coordinate of any point in the set on the streaming axis. Any operator that works on a point, its neighbors and the neighbors of its neighbors can safely operate as long as as no other operator or thread is active within the ring one reference limits.

2.3.2.9 factory

The factory used in the stream_process system is a pooling factory variant. It is related to the abstract factory and factory method design patterns described in [Gamma et al., 1994]. Different stream-operator chain configurations dynamically define sets of data member variables used at run-time that represent a point and its attributes. Hence the different forms of runtime structures are not subclasses of a common base class representing the point data, but instead are simply of type stream_data with dynamically allocated memory to hold the specified data member fields. The factory class provides the methods to allocate blocks of memory for stream_data objects. For efficiency reasons, this is implemented using memory pooling, see Section 2.3.3. The factory class uses a stream_structure instance for each stream to collect information on the attribute member fields. It computes and stores the offsets for each member variable, and sets them in the

stream operators and chain operators instances during the setup stage. The factory creates and manages a pool of appropriate stream_data objects for each stream of the current processing pipeline.

2.3.2.10 Setup Stage

To initialize a stream-operator pipeline at run-time, a setup stage is carried out to allocate, initialize and configure the operators and to collect information on the required dynamic member fields. This setting up of the system is a multi-stage process.

In the first stage of the setup process, the read operator reads the header file of the input data set, and stores the input metadata for each stream. This includes meta-data on all attributes of each stream, such as position or color for the vertex stream or vertex indices for a face or polygon stream. Then, each stream_operator is processed in sequence of the operator chain and conveys its input attribute requirements using the read() method and its outputs using the write() method.

This generates an appropriate attribute object that will be registered with the stream_structure for the respective stream, or will throw an error if an input attribute is not available. The application will then exit with an error message detailing the problem which of the input requirements of an operator cannot be fulfilled.

Additionally, some configuration settings such as the number of neighbors can be negotiated between operators.

In the last stage, the final value of the negotiated settings are passed to each operator, all other settings are finalized, and the factory computes the size of the stream_data objects for each stream, and stores the offsets in all attribute accessors in the various stream operators and chain operators.

2.3.2.11 Access Stage

At run-time, the stream_data can be accessed using attribute_accessor objects. Since each attribute is templatized with a type, this allows type-safe access to the stored variable at the cost of one (inlined) function call and a reinterpret_cast<>(). This works very efficiently and offers an additional benefit to the programmer: The types of all attribute fields are specified in the header of the respective stream-operator definition. Hence trying to access a field using a wrong type is detected at compile-time, and not only during run-time as it would be the case with other solutions (see also Section 2.3.2.1).

2.3.3 Memory Management

Pools of objects are used where possible to optimize performance by preventing continuous construction and destruction of objects. Currently, memory pools are used throughout the stream processing system, e.g. for stream_data objects, for the nodes of the spatial tree in the neighborhood operator and for reeb graph nodes. At first, the pools are implemented using the template pool class from the boost pool library [boo, 2007]. Performance testing revealed some serious problems with long-running boost pools, so it was replaced with a custom solution that improved runtime performance by 20%.

2.3.4 Parallelization and Multi-threading

Parallelization

As modern computers have evolved to contain multiple processors with multiple cores each, parallelization of the workload has become an increasing concern. Important decisions regarding the design of multi-threaded systems are the transparency and granularity of parallelization.

Transparency

Transparency measures how much a single part of the complete system needs to be aware of multi-threading issues. We chose an almost fully transparent implementation. The multi-threading functionality is encapsulated in only some parts of the system. Each operator can be programmed as if the whole system were single-threaded, as the parallelization in the stream_process system is essentially hidden from the operators. There is no need to take care of any of the complex multi-threaded programming issues such as access control or synchronization, or having to use mutexes or condition variables when implementing an stream_operator. This is achieved by implementing parallelization as follows: stream operators that can be multi-threaded are cloned, and multiple instances of multiple operators can process different slices of data in parallel.

Granularity

Per-operator multi-threading makes the system coarse-grained with respect to parallelization, but as shown above, this approach has many advantages. While it might be possible to develop a more efficient system by sacrificing transparency, the coarse grain does benefit from the fact that there are less decision points and checks required for enabling parallelization, and thus the danger of race conditions or deadlocks is reduced.

Implementation

The system needs to be able to determine if the algorithm encapsulated in an operator can be run in parallel, and how to organize its access to slices. Therefore, each stream_operator stores a boolean variables called is_multi_threadable. chain operators cannot be multi-threaded at all, as their intention is to be a global structure with respect to the active set.

is_multi_threadable

Some operators are inherently multi-threadable, while others cannot easily be parallelized, depending on their internal state. stream operators whose state for each stream_data element depends on others, such as neighborhood detection with a spatial tree data structure to store each element in its range cannot be multi-threaded easily. With the cloning approach described above, each operation on the tree would have to be synchronized between the various cloned instances. Therefore, for operators that have quasi-global structures, the is_multi_threadable flag is set to *false*. For other stream operators such as radius computation where there is no persistent state and only the current stream_data and its neighbors are read, the is_multi_threadable flag can be set to *true*.

Thread pools

The stream processing system uses thread pools to manage the different threads, based on the boost threadpool library [boo, 2007].

2.3.5 Multiple Streams

The adaptation of the stream_process technology to polygonal meshes as described in Chapter 4 has been ported back into the original stream processing system.

An second read operator (see Section 2.5.1) has been written that loads the face stream, add pointers to the face vertices into the face stream_data and adapts the reference limits (see Section 2.3.2.8).

The write operators was extended with functionality to reindex the faces, as some vertices might have been deleted from the stream, causing the numbering for all subsequent vertices to change.

More details on the streaming of polygonal meshes are given in Chapter 4.

2.3.6 Templatization

The stream processing system uses C++ templates in order to decrease the implementation effort required and allow the reuse of already written code. Because of templatization, it is possible to run the whole processing chain in either single-precision or $float$ mode, double-precision or $double$ mode or a mixed mode, where most computations are done in $floats$, but some operations known to be very sensitive to precision are performed using $doubles$. This is all done using the same source code, without having to write different versions of each operator. Other advantages that templates allow are increased configurability. As an example, the neighbor operator is templatized with the requested tree data structure, without the necessity for checks during processing or virtual function calls.

2.4 Preprocessing

2.4.1 Sorting

As for sweep-plane algorithms in computational geometry [de Berg et al., 1997], our stream-processing framework requires the points to be ordered along a spatial direction. In principle, any direction could be used as for example any of the three principal axis of the point data's modeling coordinate system. However, it is typically advantageous to align the data such that the sweep-plane intersection with the object exhibits a smaller outline. Hence for objects with a biased spatial extend, the sweep direction should be aligned accordingly.

Sorting in the direction of the longest axis of a data-aligned tight bounding box can efficiently be achieved in two phases as follows. In the first linear pass over the data points $\mathbf{p}_1, \ldots, \mathbf{p}_n \in \mathbb{R}^3$, a generic homogeneous covariance $\widehat{\mathbf{M}} = \sum_{i=1}^{n} \hat{\mathbf{p}}_i \cdot \hat{\mathbf{p}}_i^{\mathsf{T}}$ and center of mass of the points $\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{p}_i$ are accumulated, with $\hat{\mathbf{p}}$ denoting the homogeneous coordinate extension of $\mathbf{p}$. As shown in [Pajarola, 2003; Pajarola et al., 2004] this allows us to express and post-compute the actual covariance matrix $\mathbf{M} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{p}_i - \mathbf{c}) \cdot (\mathbf{p}_i - \mathbf{c})^{\mathsf{T}}$ elegantly and efficiently in homogeneous space by $\mathbf{M} = \frac{1}{n}\mathbf{T}(-\mathbf{c}) \cdot \widehat{\mathbf{M}} \cdot \mathbf{T}^{\mathsf{T}}(-\mathbf{c})$ with $\mathbf{T}(-\mathbf{c})$ being the translation matrix moving the center of mass $\mathbf{c}$ to the origin. The sorting axis is now given by the eigenvector $\mathbf{v}$ corresponding to the largest eigenvalue of $\mathbf{M}$. In the second phase, the points are transformed and sorted along their projection onto $\mathbf{v}$.

Instead of applying the above transformation and sorting offline as in [Pajarola, 2005], we have integrated it into the stream-processing framework as an online preprocess. For this purpose we have implemented an efficient out-of-core sorting algorithm based on the radix-sort technique. Besides supporting out-of-core sorting on very large data, a major goal of the sorting preprocess was to

provide an incrementally growing sorted stream of data. With radix-sort we can choose to inspect the next bit of one partition first, in a depth-first way, before continuing work on the other partition. Hence sorting can complete and progress from one end of the data range to the other. For that reason it was chosen over other out-of-core sorting algorithms. In fact, the sorting exploits a combination of radix- and insertion-sort. The data is partially sorted using radix-sort into small partitions which are then sorted using insertion-sort for performance reasons.

Taking advantage of this progressive online sorting approach, the stream-processing pipeline can be fed with the early available sorted data partitions. Thus point processing operations, e.g. such as the costly neighborhood search, can be overlapped with the sorting phase and can start with only minimal latency before the whole data has been preprocessed.

Furthermore, in order to utilize the common availability of multiple cores, the sorting preprocess was not only integrated into the main stream processor, but also adapted to use multiple threads for sorting. This not only improves performance of the online preprocess but additionally simplifies the usage of the stream processor. A thread-pool based system is utilized to efficiently distribute and perform the sorting tasks.

2.4.2 Optimal Transform

The input data set for a stream_process chain is in no specific order and not aligned properly to the streaming axis. In order improve stream processing performance, it is advantageous to transform the data set so that the longest axis is the streaming direction. Two different strategies have been implemented into the stream processing system.

Axis Swap A simple yet efficient way to prepare the sorted data set for stream processing is performing an axis swap. During the first access for each input point for sorting, we determine the axis aligned bounding box. Using this information, we can simply swap the coordinates of the longest axis with the stream axis, if necessary.

Optimal Transform A more complex approach is the so-called optimal transform. We perform a covariance analysis of the data set using the Jacobi method. Using the computed eigenvectors, we can build a rotation matrix as shown in 2.1 that will align the model optimally for stream processing. We only have to take

care to select the longest eigenvector as streaming axis.

$$\mathbb{R} = \begin{bmatrix} v_{any} & 0 \\ v_{any} \cdot v_{longest} & 0 \\ v_{longest} & 0 \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} \tag{2.1}$$

This transformation of the input data set reduces memory consumption during run-time and improves the performance of the neighborhood search by reducing the size of the active set.

2.5 System Operators

This section provides an overview of the system operators in the stream processing system, that is, operators which are required in order for the framework to function that do not implement any geometric algorithms.

2.5.1 I/O Operators

The read operator Φ_R and the write operator Φ_W are the stream operators at the beginning and the end of the processing pipeline. They are responsible for reading the input data, optionally converting it to format suitable for processing and finally the resulting data to an output file. Both operators also fulfill a number of additional tasks required for the proper functioning of the processing chain that can only be performed at the beginning or the end of the chain, respectively.

Read Operator

The read operator Φ_R acts on the input stream of point data. During the setup phase, it reads and parses the data header and maps the point data input file to the input stream. Typically, this is done via memory mapping of the input file and sequential traversal through the input data. During the point processing phase, Φ_R reads the input point data and converts it to the proper format if required.

By definition, the read operator Φ_R must be the first in a chain of operators. It uses a pool of stream_data point objects as mentioned in Section 2.3.3 to efficiently create the new objects. A user-specified number of points are read into a slice, which is then used by the rest of the operators.

The read operator also transforms the input. Currently, the following operations are implemented: Conversion into a user-specified data type such as single- or double-precision floating point numbers, and removal of duplicate input points.

In the case of multiple streams, the read operators for each stream are chained. A read operator for an optional secondary face stream will read in the faces and

updated the reference_limits (see Section 2.3.2.8) so that the streaming out of dependent vertices and faces are synchronized.

Write Operator

In the setup phase, the write operator Φ_W creates and memory maps the output file. During processing, the write operator uses the deferred writing strategy described in [Pajarola, 2005] to write points out to disk and remove them from main memory as soon as this can be done safely. Φ_W shares a pool of stream_data point objects with the read operator, as indicated above, to avoid unnecessary memory allocation and deallocation overhead.

If a secondary stream with face information is present, the write operator Φ_w will perform additional tasks such as reindexing and then writing out the faces. Reindexing is necessary if duplicate vertex removal is enabled, as some vertices might have been eliminated and therefore the indices of all subsequent vertices have changed.

2.5.2 Chain I/O Operators

A special kind of stream operators are inserted into the chain transparently if any chain operators are part of the chosen processing chain. These operators are always directly after the read operator and before the write operator (see Sections 2.5.1, or 2.5.1, respectively). These special operators contain pointers to all the chain operators and will insert and remove points from them in a thread-safe way.

2.5.3 Other Operators

Various operators have been implemented for smaller tasks or to help debugging. Table 2.1 presents a short overview of the most important of those operators and their functionality.

Name	Functionality	Variants
curv_color	Sets the point's color attribute according to the curvature.	Various curvature to color mappings.
fix_normal	Uses an average of neighbor normals if normal estimation failed.	
fix_scales	Replaces the scales with neighbor averages if scale estimation failed.	
nb_store	Writes out the index of and distance to each neighbor point.	
normal_color	Sets the point's color attribute according to the normal.	Various normal to color mappings.
perturb_normal	Perturbs the normal by a user--specifiable amount.	
print_*	Prints out the values of point attributes to the console.	
stats	Collects statistics about the current processing run.	

Table 2.1: *A listing of miscellaneous non-geometric operators implemented in the stream processing system.*

2.6 Visualization

An important application of the stream processing system is the processing of a scanned model for visualization of the point data set. The visualization application for stream processed models is a native application for Mac OS X called Point Splatter. All rendering is done using OpenGL, with the help of GPU Shaders written in either GLSL, the OpenGL Shading Language, or Cg, a proprietary, multi-platform shading language developed by nVidia. A screenshot of the application is shown in Figure 2.7.

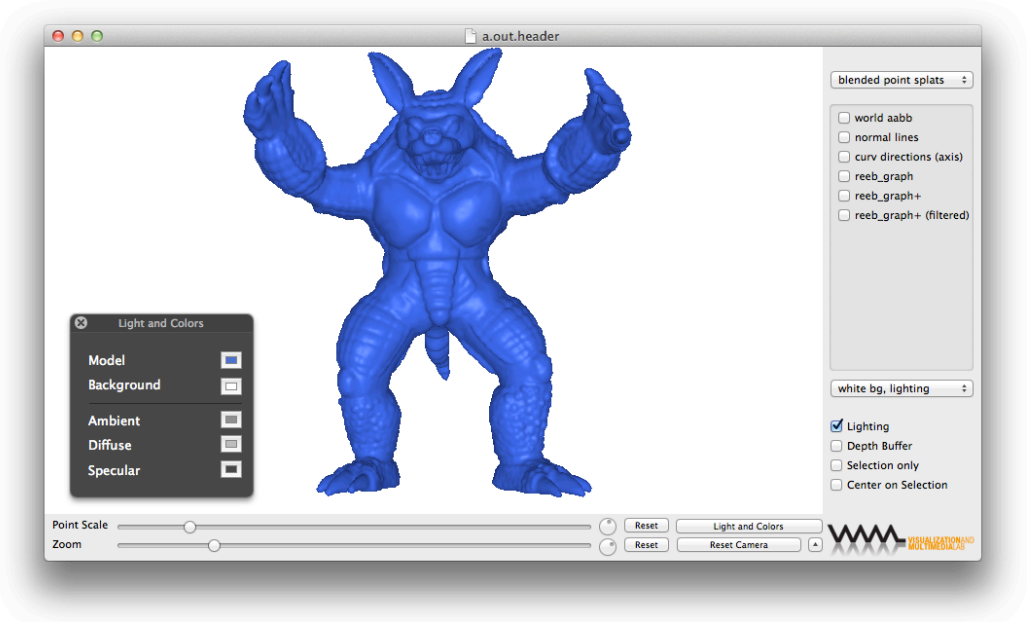


Figure 2.7: *A screenshot of the PointSplatter visualization application for the stream processor.*

The PointSplat application itself is mostly based on two custom C++ libraries, glibber library and point splat library, and a Cocoa-based graphical user interface (GUI).

2.6.1 libglibber

The glibber library is an OpenGL library written in C++ and developed at the Visualization and Multimedia Lab of the Department of Informatics at the University of Zurich. The library provides a simple, object-oriented way to use common OpenGL functionality from C++, and uses common default values for many operations. It contains classes for basic functionality such as user-controlled

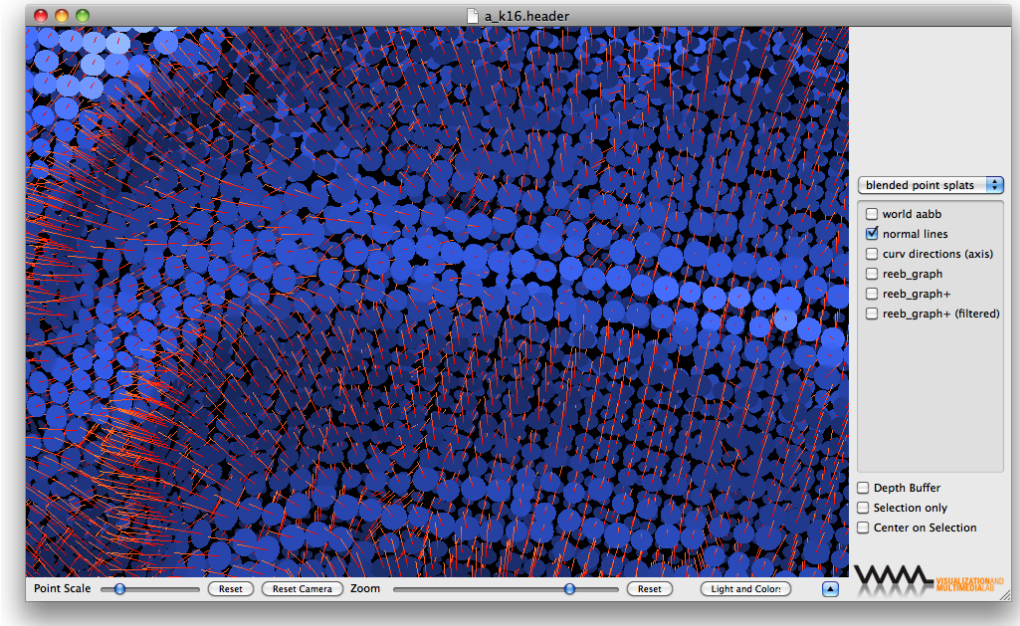


Figure 2.8: *A screenshot of the PointSplatter visualization application showing splat discs and normal lines.*

cameras and the underlying modelview- and transformation-matrices and viewport, loading and using textures and lighting. More advanced functionality includes render-to-texture classes based on different methods, classes for loading and using GPU code in the GLSL and Cg shading languages and for natural and uncomplicated setting of uniform values, and classes that simplify the handling of modern GPU rendering techniques such as vertex buffer objects, frame buffer objects and pixel buffer objects.

2.6.2 libpointsplat

The point splat library is a rendering library for OpenGL based on glibber library, written in C++, GLSL and Cg and developed at the Visualization and Multimedia Lab of the Department of Informatics at the University of Zurich. The library contains the algorithms used for visualizing a point set model. A list of the currently implemented algorithms are shown in Table 2.2. The point splat algorithms are variations on the point splat algorithm presented in [Botsch et al., 2005; Pfister et al., 2000].

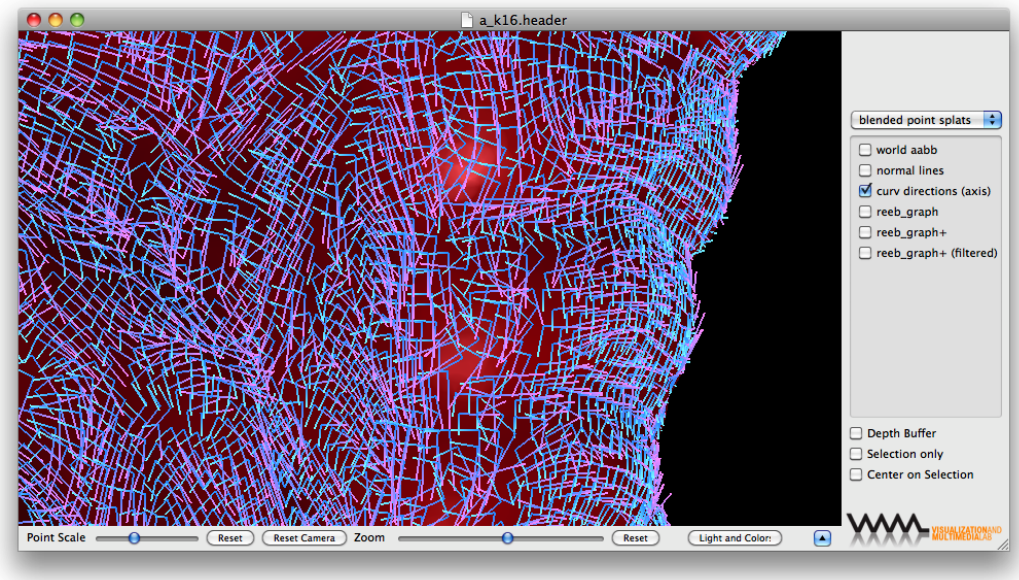


Figure 2.9: *A screenshot of the PointSplatter visualization application showing principal curvature directions as lines.*

Technology	**Result**	**Variants**
GL_POINTS	Quadrangles	
Cg Shading Language	Point splats	default, blended
	Elliptical splats	default, blended
GLSL Shading Language	Point splats	default, blended
	Elliptical splats	default, blended
	Reeb graph	default, embedded, filtered
	Normal arrows	
	Curvature direction lines	
	Normal error	default, blended, textured

Table 2.2: *The basic rendering and debug visualization algorithms currently implemented in the point splat library.*

2.6.3 PointSplatter

Point Splatter is a native Mac OS X application based on Cocoa to visualize the output files of the stream processing system. Written in part C++ and part Objective-C, it allows the user to zoom into a model, change many visualization parameters such as the render algorithm, lighting, point splat size and the default color used for rendering if the model doesn't specify its own color.

In addition, Point Splatter supports a series of operations helpful for debugging and algorithm development. Debugging helper algorithms are e.g. the normal line and curvature line renderers that draw a line onto the surface in normal direction (as shown in Figure 2.8 or the principal curvature directions as shown in Figure 2.9, respectively.

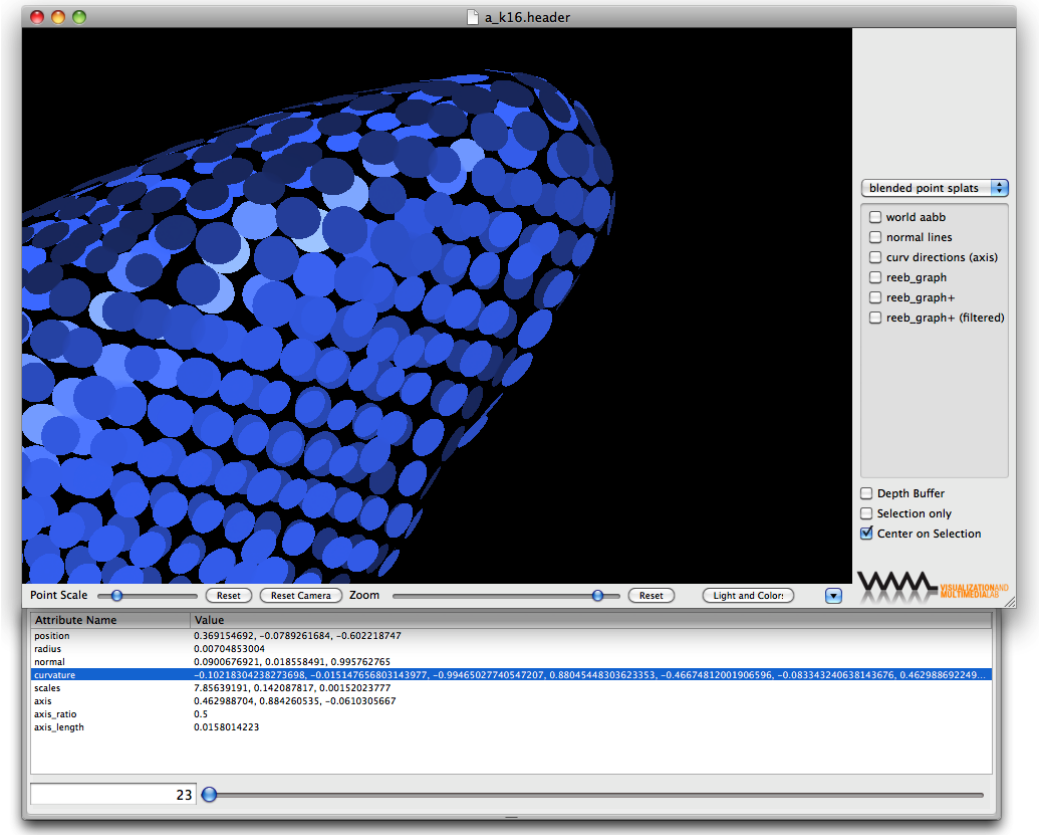


Figure 2.10: *A screenshot of the PointSplatter visualization application showing the data drawer with all vertex attributes in human readable form for the selected point.*

Any rendering algorithms can be combined, which allows to simply combine a point splat render algorithm and a debug support algorithm to gain a better

overview. Additionally, the values of all vertex attributes of a single point can be displayed in a drawer, as presented in 2.10.

The list of rendering algorithms and debugging visualizations is generated dynamically based on the selected point set model. This allows filtering of unsuitable algorithms, e.g. those that require a vertex attribute that is not available in the chosen point set.

The application is written in a way that is completely self-contained, which means that all the required libraries are included in the application bundle, and no installation or dependency management in necessary.

2.7 Results

In this section, we will present results on the runtime performance of various operators of the stream processing system as well as statistics on memory usage. All experiments were performed on the machine specified in Appendix A.

Memory Usage

One of the main goals of the stream processing system is the processing of large data sets. We tested the suitability of the system for this purpose by recording the maximum active set for a full processing chain[2]. The active-set size measurements were performed in single-threaded mode, with $k = 8$ nearest neighbors and a slice packet size[3] of $s = 1000$. The results for various point models are presented in Table 2.3.

Model	Number of Points	Max. Active Set	In Percent
Bunny	35'947	17'000	47.29%
Armadillo	172'982	38'000	21.97%
David Head	2'000'651	128'000	6.40%
Dragon	3'609'600	643'000	17.81%
David 2mm	4'129'614	197'998	4.80 %
Statuette	4'999'996	328'000	6.56 %
Lucy	14'027'872	324'967	2.32 %
David 1mm	28'184'526	592'953	2.10 %
St. Matthew	186'836'670	2'122'000	1.14 %
Atlas	254'837'035	1'240'999	0.49 %

Table 2.3: *Overview of the maximum active set sizes when processing a full chain (see text). All models were processed with a neighborhood of $k = 8$ and a slice-size of 1000. While some of the smaller models have a large percentage of actual points in memory, this quickly falls with growing model sizes. All models with 10 million or more points never have more than 2.5% of the total model size in memory, and the largest model, Atlas, has less than half of a percent of total points in memory at any time.*

We can see that for the smaller models, a large subset of the model is loaded into memory during processing, but this quickly decreases with increasing model size. None of the tested models with 10 million points or more exceed an active set size of 2.5%, and for the largest model, no more than 0.49% of the total model

[2] Table 2.4 presents a list and a short overview of each operators used for the full processing chain.

[3] Some active set sizes that are not a multiple of the slice size. This is due to the duplicate vertex removal performed in the read operator.

has to be loaded into memory at any point. Reducing the slice size decreases the maximum size of the active set, as shown in [Bösch and Pajarola, 2008], but at the cost of increasing the run-time performance. Note also that the full chain is could be considered a worst-case example, as it consists of 14 individual operators.

Operator	Function
read	read input
chain-in	chain operator helper
neighbor	detects neighborhood
radius	computes radius
covariance	computes covariance
normal	estimates surface normal
flip_normal	normal orientation
curvature	estimates curvature directions and scales
splat	computes elliptical splat
apss	fits a sphere at point position
curv_scale	scales curvature to the actual value
gauss	determines the sign of the Gaussian curvature
chain-out	chain operator helper
write	write out result

Table 2.4: *The operators used for the full processing chain. More information on the individual operators can be found in Section 3.*

.

Run-time Performance

Data on the runtime performance of individual operators is presented in their respective sections in Chapter 3. Table 2.5 presents the processing time for the full processing chain on large models with $k = 8$ neighbors. The experiments were run on the machine specified in Appendix A, with multi-threading enabled with a maximum of four threads.

The Effect of Multi-Threading

The parallelization of the stream processing system was tested by comparing runs of the full processing chain[4] in single-threaded mode and with 2, 4 and 8 threads. In Table 2.6, the the total processing time run-times are shown, Figure 2.11 presents the processing time per point. As comparison, the time for the neighborhood operator is shown, as neighborhood search is the most demanding

[4] See Table 2.4

Model	Number of Points	Full Chain
Lucy	14.02 Mio	118.5s
David 1mm	28.18 Mio	226.91
St. Matthew	186.83 Mio	3462.63s
Atlas	254.83 Mio	5438.31s

Table 2.5: *Run-times for the full processing chain on large models, in multi-threaded mode, with $k = 8$ neighbors and a maximum of 4 threads.*

Model	Single-threaded	2 Threads	4 Threads	8 Threads
Bunny	0.24s	0.21s	0.21s	0.33s
Armadillo	1.70s	1.11s	0.92s	0.95s
Happy Buddha	5.75s	3.80s	3.29s	3.37s
David Head	20.84s	14.51s	12.57s	12.67s
Dragon	50.79s	39.74s	36.60s	36.96s
David 2mm	41.14s	26.68s	21.55s	22.05s
Statuette	66.57s	50.03s	45.72s	46.31s

Table 2.6: *A comparison of the running times for the full chain in single-threaded mode and in multi-threaded mode with 2, 4 and 8 threads. There is little difference between the 4- and 8-threaded results, but both are significantly faster than both the single-threaded and the 2-threads run.*

non-multithreadable operator and represents the minimal baseline even an ideally parallelized system could not beat. We can see that the single-threaded run takes significantly longer than the multi-threaded runs, and that 4 threads beat 2 threads in every case. The version with 4 threads only takes between 20% and 30% more time than the neighborhood search alone except for the bunny case, where the model is probably too small. With a processing time of less than $\frac{1}{4}th$ of a second, it seems that the overhead when managing multiple threads is too large.

Summary

To summarize, we present an efficient system for the streaming of large data sets that is configurable at run-time, with only a small percentage of the complete model loaded into memory at any time. The concept of stream operators allows the encapsulation of algorithms, while allowing access to all required point attributes, and the reference limits guarantee that no read and write operations overlap in multi-threaded mode. Run-time structures enable an operator devel-

oper to read and write arbitrary point attributes in a way that is efficient as well as type-safe. The parallelization is shown to be efficient, with no more than 30% higher run-times of the full processing chain compared to the run-time of the largest non-parallelizable operator on most models.

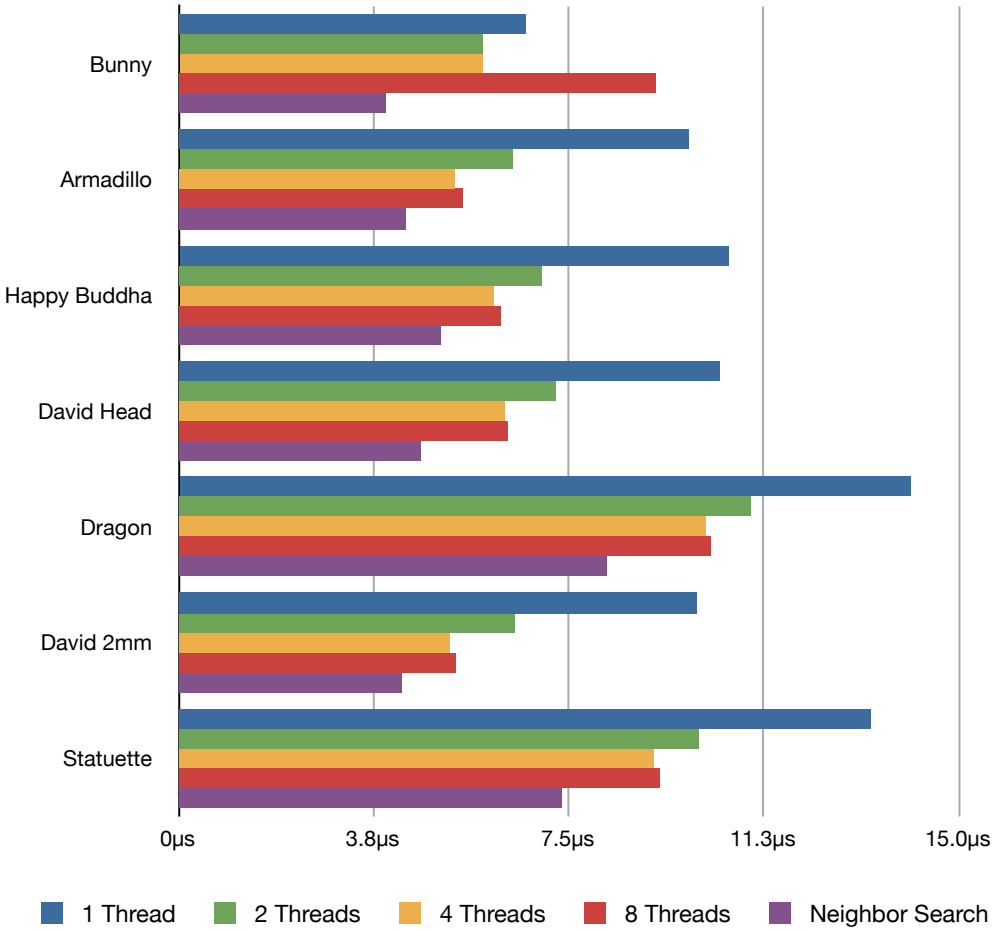
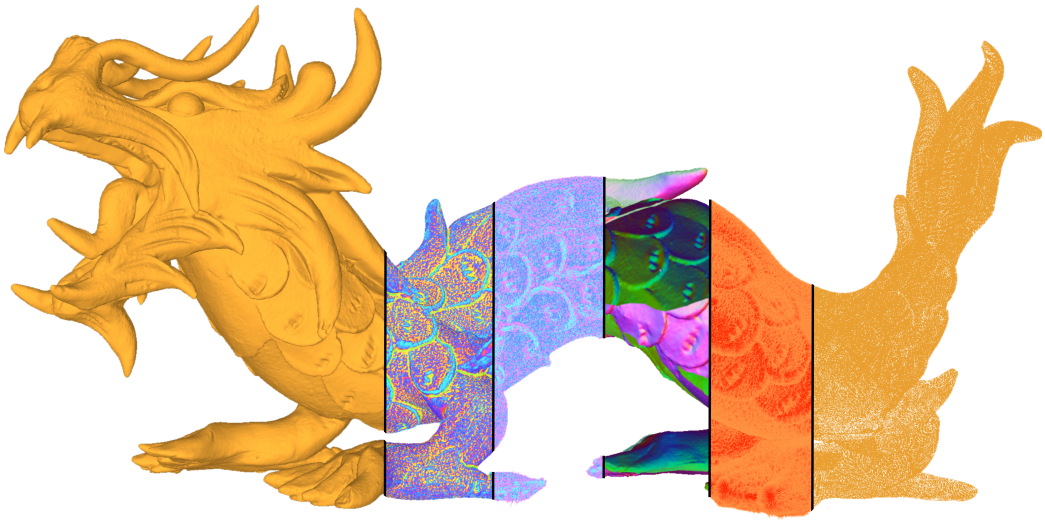


Figure 2.11: *A comparison of the multi-threading performance of the stream processing system. The full processing chain was run in single-threaded mode and with 2, 4 and 8 threads. Displayed is the full processing time, averaged over 3 runs, divided by the number of points. For context, the time required for neighborhood search, the most demanding non-multithreadable operator, is shown. We can see that in the best case we get a significant speed-up, and the total processing time is only about 20-30% higher than the neighborhood search.*

CHAPTER 3

POINT GEOMETRY



3.1 Overview

In this section, the actual geometric processing implemented in the stream processing system is presented. We begin with an overview of geometric point processing and stream processing in general, followed by an in-depth description of the background, theory and algorithms of each of the various geometric operators. A short explanation of implementation details is then concluded by a presentation and discussion of the results obtained with the operator.

3.2 Background

Points as 3D surface modeling and rendering primitives have been introduced as early as in [Levoy and Whitted, 1985] and [Grossman and Dally, 1998]. A number of efficient hardware supported rendering algorithms such as [Rusinkiewicz and Levoy, 2000; Ren et al., 2002; Botsch et al., 2002; Botsch and Kobbelt, 2003; Pajarola et al., 2004] have been proposed and subsequently further improved. The fundamental theory and algorithms for point-based modeling and rendering are described in [Gross and Pfister, 2007b], and surveys on point-based rendering (PBR) have been presented in [Sainz et al., 2004; Sainz and Pajarola, 2004] and [Kobbelt and Botsch, 2004]. Apart from rendering which has been well studied and extended to out-of-core [Rusinkiewicz and Levoy, 2001; Gobbetti and Marton, 2004; Pajarola et al., 2005] or transparent rendering [Zhang and Pajarola, 2006b; Zhang and Pajarola, 2007], low-level geometry processing techniques for point data have been discussed in [Pauly and Gross, 2001; Liu et al., 2002; Pauly et al., 2003; Jones et al., 2004; Weyrich et al., 2004]. However, these methods are aimed at processing only moderately sized point sets that fit into main memory.

Sequential organization of point data has been addressed specifically for rendering and network transmission in [Dachsbacher et al., 2003; Pajarola et al., 2005] and [Rusinkiewicz and Levoy, 2001]. More general low-level geometric operations are applied to a stream of points in [Pajarola, 2005], which we will discuss in the following section.

Streaming has chiefly been used in processing digital audio and video data which in contrast to 3D geometry is inherently sequentially organized, i.e. in time. The sweep-line concept in geometry processing [de Berg et al., 1997] is conceptually closer than multimedia streaming, since our basic stream-processing follows a similar idea of sweeping a plane over the point cloud data. In the context of 3D geometry, streaming has been introduced for simplification and compression operations on polygonal meshes [Isenburg et al., 2003; Wu and Kobbelt, 2003; Isenburg et al., 2005; Vo et al., 2007], which generally grow and process mesh regions sequentially in an order that limits main memory usage. Specifically

for rendering, a streaming mesh layout has been proposed in [Isenburg and Lindstrom, 2005]. These streaming approaches on meshes, however, do not support low-level geometry processing operations, and more importantly, do not directly apply to raw point data processing as mesh connectivity is required.

Recently, different streaming frameworks for surface reconstruction from points have been presented. [Cuccuru et al., 2009] propose a parallel MLS approach and [Bolitho et al., 2007] a Poisson-based multi-resolution streaming framework based on a sparse octree. In [Isenburg et al., 2006] and [Allegre et al., 2007] slice-based streaming algorithms are proposed for Delaunay triangulations. There, the space is partitioned into explicit regions as opposed to the implicit region partitioning used in this paper by using a sweep plane.

Finally, in graphics the concept of streaming images and geometry data has been used in the context of remote rendering where 3D data is to be displayed on a remote display (e.g. [Engel et al., 2000], [Noimark and Cohen-Or, 2003] or [Cheng et al., 2004]). Again, low-level data processing is not the focus in these approaches but the network transmission of data to a remote device.

3.2.1 Point Models

In the following sections, results of the various algorithms are presented using the models listed in Table 3.1. The unprocessed size is the size[1] of the input model in single (float) precision, including any point attributes available in the original source, but without face information. The processing size is the size in memory required for running a full processing chain including normal, curvature and splat estimation. Note that this is only the size of the actual point attributes, without global data such as the neighborhood tree structure or the memory required for the operators. As we can see, larger models could not be processed efficiently without using streaming techniques as the size required is much larger than available memory.

[1] We use the traditional definitions for MB and GB, meaning 1024^2 and 1024^3, respectively. Therefore file sizes may appear larger on your hard-disk than the size listed here.

Model Name	Number of Points	Size Unprocessed	Size in Memory
Stanford Bunny	35’947	0.718 MB	7.23 MB
Armadillo	172’974	2.076 MB	33.5 MB
Happy Buddha	543’652	6.523 MB	105.25 MB
David Head	2’000’651	32.01 MB	394.95 MB
Dragon	3’609’600	43.315 MB	698.98 MB
David (2mm)	4’129’614	66.073 MB	815.23 MB
Statuette	4’999’996	60 MB	967.98 MB
Lucy	14’027’872	168.334 MB	2.715 GB
David (1mm)	28’184’526	450.952 MB	5.563 GB
St. Matthew	186’836’670	2.989 GB	36.02 GB
Atlas	254’837’035	4.077 GB	49.13 GB

Table 3.1: *Details about the different point models used for testing the various algorithms in the* stream processing system.

3.3 Neighborhood

3.3.1 Background

Stream processing operators work on the local geometry of a point, which is defined as the current point itself and a neighborhood $\mathcal{N}$ of spatially close points. $\mathcal{N}_i$ of a point p_i of a set of points p_0, p_1, ... p_n is defined as the subset of n points p_{j0}, p_{j1}, ... p_{jn} with a minimal distance to p_i. Efficient determination of $\mathcal{N}_i$ is a difficult problem [Samet, 2006].

The stream_process system uses multi-dimensional spatial tree data structures to organize neighborhood detection. Spatial partitioning methods such as trees allow to perform neighborhood queries in a much more efficient way than other approaches such as linear search. A spatial tree consists of nodes partitioning the space comprised by the data set. Each (complete) level of such a tree spans the whole data set, and each parent node contains the space of all of its child nodes.

A naive approach to finding the nearest neighbor for a certain point p_i is to compute the distances between all the points and then selecting the point the minimal distance. The runtime complexity of such an approach is O(N), and O(N^2) for finding the nearest neighbor for each point p_i.

Using a spatial data structures improves on this. Each point p_i is inserted into the tree according to the tree's criteria. When searching for the nearest point, the tree can be traversed to find the node that p_i was inserted into. Starting from this node, we can build a priority queue of nodes to be searched. This reduces the runtime complexity to logarithmic order. A more detailed description of the kNN search algorithm can be found in section 3.3.2.1, and more information on various multi-dimensional data structures can be found in [Samet, 2006].

3.3.2 Algorithms

This section presents the basic neighborhood detection algorithm and the different tree data structures implemented for use with this algorithm.

3.3.2.1 k-Nearest Neighborhood Search

In the stream processing system, we use a generic k-nearest neighbor algorithm similar to the one described in [Samet, 2006]. Search starts at the tree node that the current point is in. Two min-priority queues are used, the first one stores pointers to the k-nearest neighbor candidates, and the second one contains nodes close to the current node. We will call those two queues the node-queue and the candidate-queue.

To begin the search, the current node is selected, and all points in it are added to the candidate-queue, with the squared distance[2] used as priority. We keep the candidate-queue at a maximum size of k, and remove any point that has a distance larger than that of the k-th neighbor. Once the current node is completed, we traverse the tree to collect neighboring nodes, which are added to the node-queue using the minimal distance from p_i the node[3] as priority indicator in the priority queue. We then pop the closest node from the node-queue, select it as current node and repeat the procedure.

The search expands to new nodes as long as the minimum distance to the top node in the node-queue is smaller than the distance to the k-th nearest neighbor candidate. As soon as this distance limit is reached, it is impossible for any point in those nodes to be closer to the current point as the k-th neighbor candidate we already have, and we can safely stop the search.

3.3.2.2 kd-Tree

A kd-Tree [Bentley, 1975] is a spatial binary tree. The k in kd-tree denotes the number of spatial dimensions that the tree encompasses. Each level is a subdivision of the parent space in one axis. Figure 3.1 presents an example of a kd-tree. In the general case, the split axes are iterated over when traversing the tree, so the children of the root node might subdivide the point set according to a coordinate on the x-axis, on the next level according to the value of the y-coordinate and so on, cycling through all k axes. Common values for the split value on the current axis are the median of all points contained in the node, or some approximations to avoid median computation.

Choice of Split Value

As the stream processor continually inserts and removes points from the tree, a suboptimal choice of split value in higher levels can lead to degeneration of a whole subtree. While this can be fixed with various rebalancing approaches, a simple yet efficient choice is to set the split value to always lie on the center of the parent nodes' spanned distance on the subdivision-axis. This version of the kd-tree is called point region or pr-kd-tree according to Samet [Samet, 2006].

Bucketing

There are different strategies for storing the actual payload data in the tree. The most common one is storing one point in each node. This has the disadvantage

[2]The squared distance is used to avoid unnecessary but expensive square root operations.

[3]Or, more exactly, we use the minimal distance from the current point p_i to the bounding box of the respective tree node.

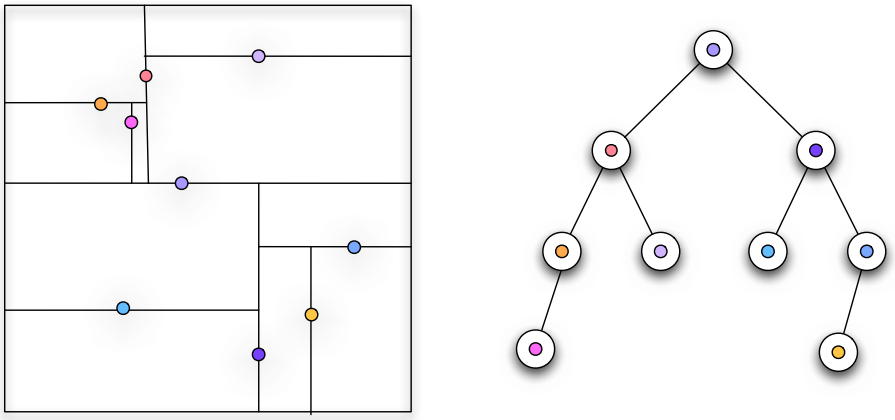


Figure 3.1: *A point kd-tree. The irregular shape is a result of the chosen space subdivision strategy.*

that for larger point sets, the tree quickly becomes very deep. Bucketing can reduce this effect by using containers called buckets in the leaf nodes. Each of these buckets contains multiple points, and only leaf nodes store actual points. Region trees are usually implemented with buckets, as otherwise, they can quickly become very deep, since the split axis of the different levels are not guaranteed and in most cases won't split the points. Technically, bucketed point region kd-trees are a type of trie, but this name is rarely used in practice.

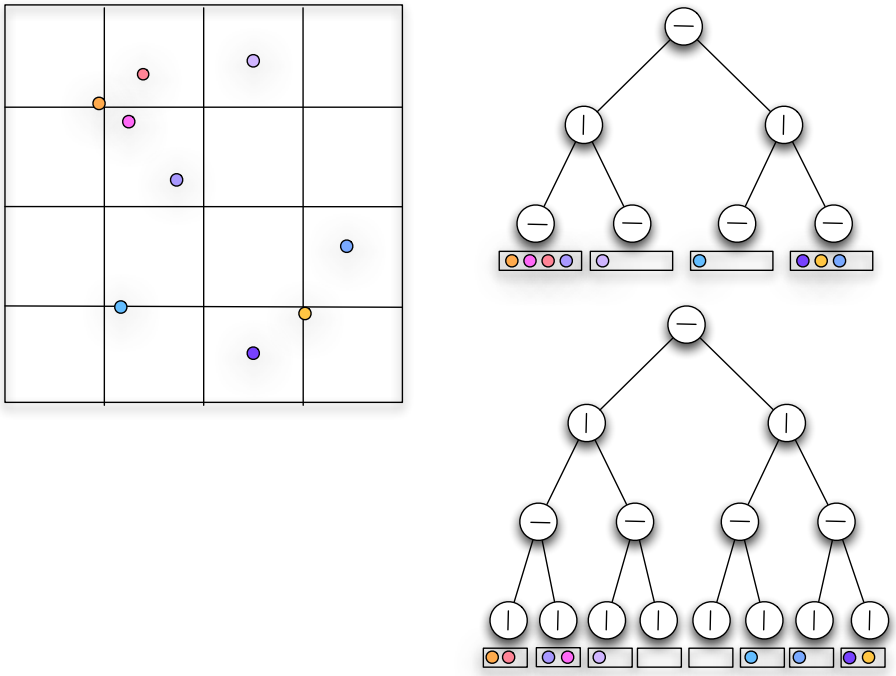


Figure 3.2: *A bucketed point region kd-tree. To the right, the tree structure is shown for buckets of size 2 and 4. Note how increasing the bucket size decreases the tree depth. As this is a point-region kd-tree, the subdivision of the space is regular and resembles a quadtree.*

3.3.2.3 Quad- and Octree

Quad- and Octrees are the two- and three-dimensional version of the same tree. They are not binary trees, instead each node has four (quadtree) or eight (octree) children that subdivide the space contained by the parent regularly, as shown in Figure 3.3. As with the point region kd-tree, buckets are used to avoid growing the tree unnecessarily deep, and the data structure should theoretically be called bucketed point region quadtrie or octrie.

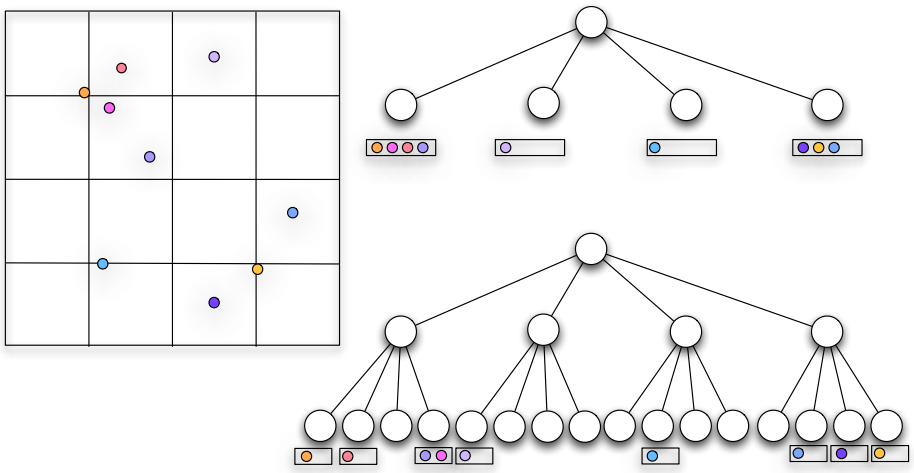


Figure 3.3: *A bucketed quadtree. To the right, the tree structure is shown for buckets of size 2 and 4. Note how increasing the bucket size decreases the tree depth.*

3.3.3 Implementation

The neighborhood operator is implemented as a set of templatized classes such that different tree structures can be used without having to change the tree code. Both a chain_operator as well as a stream_operator version of the neighborhood operator is available.

Additional Operators

An additional operator that enhances the functionality of the neighbor operator is the nb_store operator, a helper operator for neighborhood detection. If enabled, it writes out the indices of the neighboring vertices as well as the squared distance to each neighbor to the output data file.

3.3.4 Results

This section presents the qualitative and quantitative results of the different tree structures evaluated for neighborhood detection.

Neighborhood Estimation Quality

The neighborhood detection approach we used computes an approximation of the true nearest neighbor set. The detected neighborhoods using the different tree data structures are to the largest part identical. The differences affected far fewer

than 1% of all points. Table 3.2 shows the maximal amount of different neighbors found for a neighborhood size k of 8 for any of the data structures.

Model	Different Neighbors	Difference in Percent
Bunny	1	0.000348%
Armadillo	3	0.000216%
Happy Buddha	259	0.005955%
David Head	17	0.000106%
Dragon	20	0.000069%

Table 3.2: *The maximal number of different estimated neighbors using any of the implemented tree data structures.*

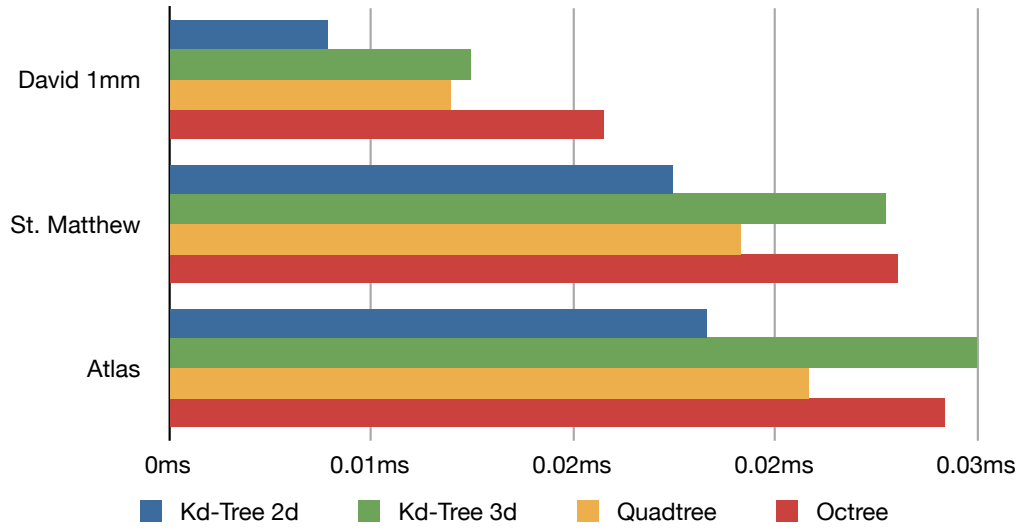


Figure 3.4: *Comparison of the performance of different tree data structures in the neighborhood operator for large point data sets. The time is the total processing time in single-threaded mode, averaged over five runs, divided by the total number of points, using a neighborhood size of k=8. The processing chain used for this test was a neighbor-only chain, that is, a chain consisting only of read and write operators and the respective neighbor operator.*

Performance

To measure the run-time performance of the different tree data structures, the models were processed using a neighbor-only chain, that is, a processing chain

consisting only of the read, neighbor and write operators. As shown in Figure 3.4 for large and Figure 3.5 for small models, the 2D variants of the tree data structures outperformed the 3D variants by a large margin, and the kd-trees performed better than the quadtree/octree variants.

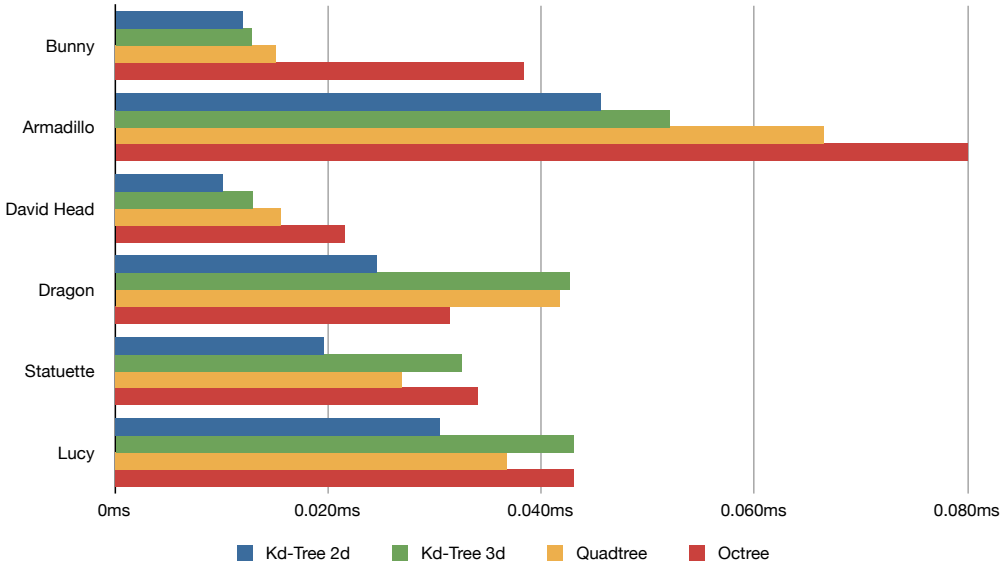


Figure 3.5: *Comparison of the performance of different tree data structures in the neighborhood operator for small point data sets. The time shown is the total processing time in single-threaded mode, averaged over five runs, divided by the total number of points, averaged for neighborhood sizes of k of 8, 16, 32 and 64.*

The advantage of the 2D trees can be explained as follows: Since the points are streamed into and out of the active_set in streaming direction, high-level streaming-axis subdivisions lead to an effect where, at the beginning, all points lie in direction of the smaller streaming-axis values. The additional subdivision can only increase search performance when most of the points in the enclosed space have been streamed in, but before a large number of them have already been streamed out. This, in turn, depends on the structure of the point model, mainly the distribution of points and their density in subregions, which can explain the differences in relative performance between the tested point data sets.

The reason for the performance advantage of the kd-tree variants over the quadtree/octree versions is likely a combination of additional float comparisons, the bucket approach and the limitation on maximal tree depth.

3.4 Radius

3.4.1 Background

One goal of the stream_process system is to compute the attributes required for visualizing a model, computing a visual representation of the point data set. To achieve this, we compute a radius for each point p_i. This radius and the surface normal allow the rendering of point splats or discs that optimally cover the whole surface. Two algorithms for computing the radius have been implemented. A visual overview of the two approaches is presented in Figure 3.6.

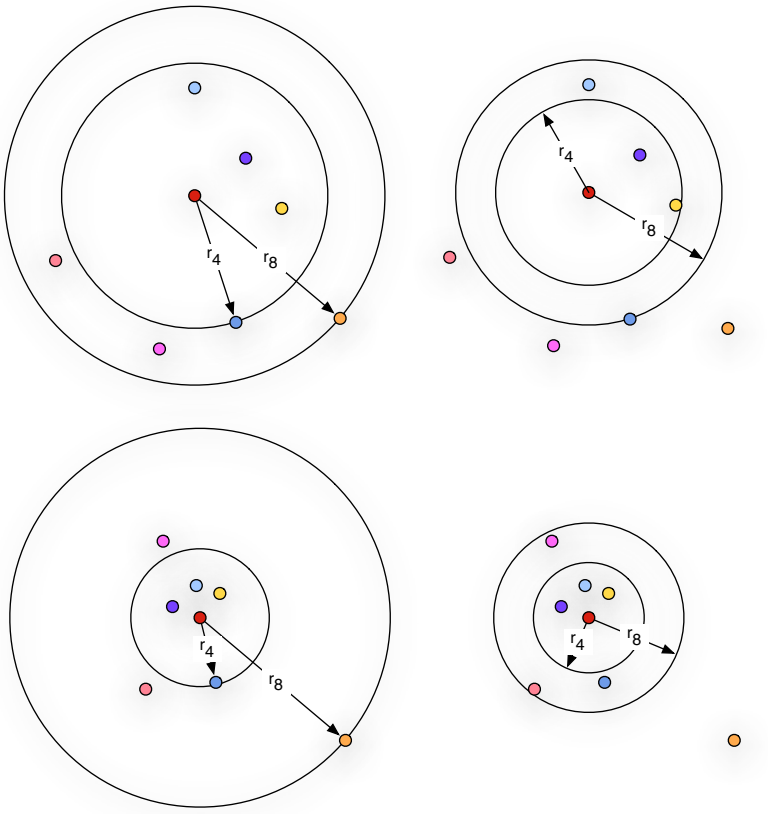


Figure 3.6: *The radius of a point. The k-th-nearest-neighbor distance radius on the left is directly dependent on the user-specified number of neighbor, 4 or 8 in the figure, while the MLS radius algorithm shown on the right uses the point density, an approach much more geometrically justifiable and less dependent on arbitrary parameters.*

3.4.2 Algorithms

A short overview of the two algorithms used in the stream processing system is given below.

k-th Neighbor Distance

A very simple approach to computing the radius of a point is to take the distance to its farthest neighbor as shown in 3.1. While computationally inexpensive and straight-forward, it has a large disadvantage in being completely dependent on the user-specified k, the size of neighborhood $\mathcal{N}_i$, and not on any inherent geometric criteria. Figure 3.6a shows how a single outlying point can distort the radius.

$$r\_kth_{p_i} = \forall p_j \in \mathcal{N} : max(||p_j - p_i||) \tag{3.1}$$

Local Sampling Density Radius

A better approximation of the area covered by the point's disc is the mean least squares or MLS support radius. We consider the point density in the point's neighborhood to derive an improved radius estimate. If there are k points within a certain area, each of the points should cover the $1/k$-th part of the area. Variable b denotes the boundary points.

$$b = \sqrt{\pi} * (\sqrt{k} - \pi) \tag{3.2}$$

$$r\_mls_{p_i} = 2 * \sqrt{r\_kth_{p_i}/(k - b)} \tag{3.3}$$

Using the density should make the computed radius value much less dependent on the user-specified value k, the size for the neighborhood $\mathcal{N}_i$, and more the actual geometry of the model.

3.4.3 Results

A processing chain with the radius operator was run multiple times, with different sizes k of the neighborhood. The results are presented in Figure 3.7. We can see that the 'MLS' algorithm captures the radius of the point much more stably than k-th Neighbor distance, where the radii grow rapidly with increasing neighborhood size, as expected. An interesting observation is that the 'MLS' radius is close to the 'kth-neighbor-distance' radius for $k = 8$ neighbors for all models.

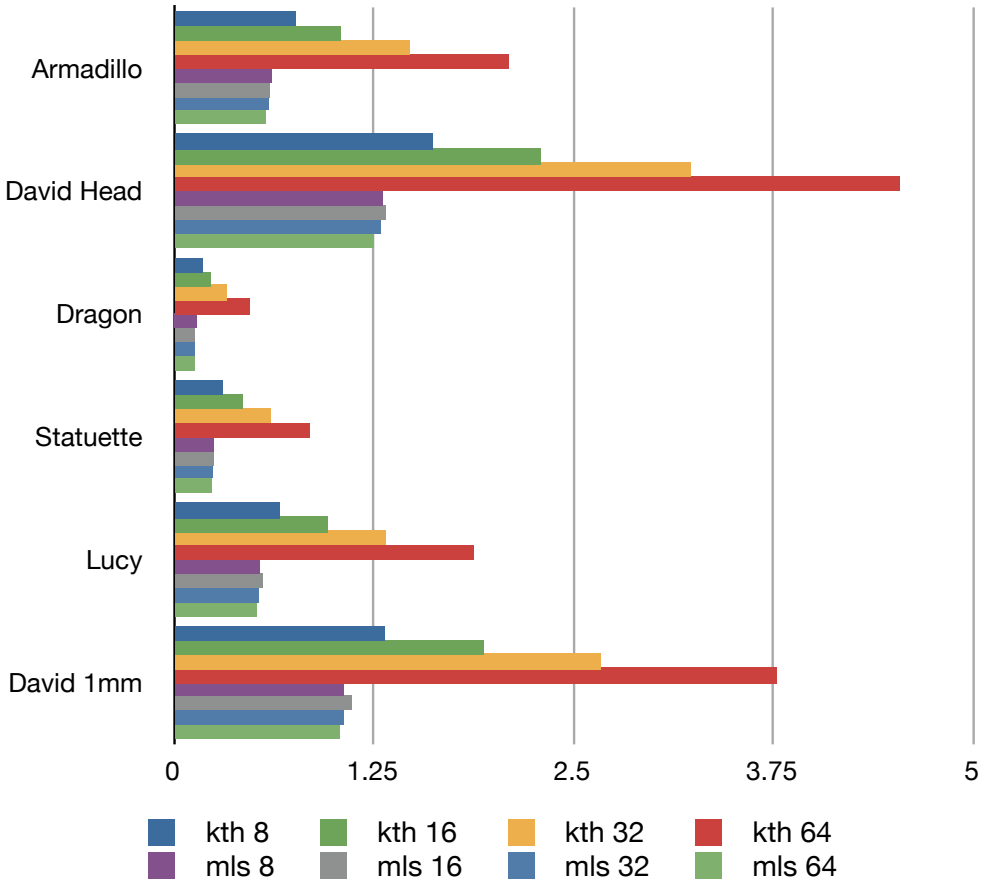


Figure 3.7: *The results of the two algorithms implemented in the radius operator. We can see that 'kth-neighbor-distance' radii grow quickly with increasing neighborhood size, while the 'mls' radii are very stable.*

3.5 Normal Estimation

3.5.1 Background

Normal estimation is one of the most important algorithms in the stream processing system. The input data usually consists of only point positions and point colors, but information on the surface orientation in the form of a normal vector is required for many subsequent algorithms and for applications such as point splat rendering.

3.5.2 Algorithms

This section presents the algorithms used to estimate or compute the surface normal. The two point-based methods, covariance-based normal estimation and the natural normals approach, are presented and discussed in detail. A third method allows to compute the vertex normals from face normals if a secondary stream with triangle mesh information is present.

3.5.2.1 Covariance-Based Normals

A widely used approach to normal estimation for points is plane fitting in different variations [Alexa et al., 2001], [Pauly et al., 2002a], [Pauly et al., 2003], [Mitra et al., 2004]. The approach presented here is based on previous work by [Pajarola, 2005].

local least squares (LLS) is used to compute a fit of a plane to a point p_i and its neighborhood $\mathcal{N}_i$. We define $\mathcal{N}_{i+}$ to comprise the set of points that contains p_i and its neighborhood $\mathcal{N}_i$.

$$\mathcal{N}_i = \{p_{j_0}, p_{j_1}, ...p_{j_k}\}. \tag{3.4}$$

$$\mathcal{N}_{i+} = \{p_i, \mathcal{N}_i\}. \tag{3.5}$$

We perform eigenvalue analysis and eigenvector decomposition of the covariance matrix M_i over $\mathcal{N}_{i+}$, and fit a plane using local least squares (LLS). A moving least squares (MLS) representation of the covariance is expressed as weighted sum similar to the approach proposed by [Alexa et al., 2001].

$$M_i = |\mathcal{N}_i|^{-1} \cdot \sum_{p_j \in \mathcal{N}_i} (p_j - p_i) \cdot (p_j - p_i)^T \cdot \theta(|p_j - p_i|). \tag{3.6}$$

The weight function $\theta(r)$ is typically defined as a smooth, radially symmetric, monotonically decreasing Gaussian function.

$$\theta(r) = e^{-r^2/2\sigma^2} \tag{3.7}$$

The variance σ^2 is adaptively defined as the local point density estimate, as first proposed by [Mitra and Nguyen, 2003].

$$\sigma^2 = \pi \cdot MAX_{p_j \in \mathcal{N}_i} \frac{|p_j - p_i|^2}{|\mathcal{N}_i|}. \tag{3.8}$$

The normal n_i of a point p_i is now computed as the normalized cross product of the two eigenvectors with the largest eigenvalues of the MLS covariance M_i.

$$x_i = v_{e_0} \times v_{e_1} \tag{3.9}$$

$$n_i = \hat{x_i} = \frac{x_i}{|x_i|} \tag{3.10}$$

3.5.2.2 Natural Normals

An alternative approach to estimating surface normals was proposed by [Alexa and Adamson, 2009], an extension of their work on point set surfaces [Alexa et al., 2001].

The basic idea behind point set surfaces is to use the points from the point data set as input for a function whose zero level set defines a surface. A locally weighted centroid $c(x)$ of the points at position x together with an approximate gradient $n(x)$ can be used to estimate the direction of the normal vector n_x at a position x. This local centroid is defined as

$$c(x) = \frac{\sum_i \theta(||x - p_i||)p_i}{\sum_i \theta(||x - p_i||)}. \tag{3.11}$$

following [Alexa and Adamson, 2009]. θ is a weight function required to be positive smoothly decaying.

We can determine the normal direction based on the centroid presented above. As shown in Equation 3.11, moving x in a tangential direction will have a large effect, while moving x in orthogonal direction has no influence on $c(x)$. Using this idea, it is possible to determine the orthogonal direction and therefore estimate the normal vector at x. The eigenvalues and eigenvectors of the Jacobian of $c(x)$

are examined and the eigenvector with the smallest corresponding eigenvalue is chosen as estimate for the normal vector.

The matrix diagonalization using the Jacobian is defined as follows

$$J_c(x) = E\Lambda E^T \tag{3.12}$$

and can also be written more verbosely as

$$J_c(x) = (e_0, e_1, ..., e_{n-1}) \begin{bmatrix} \lambda_0 & & \\ & \ddots & \\ & & \lambda n - 1 \end{bmatrix} (e_0, e_1, ..., e_{n-1})^T \tag{3.13}$$

Assume that the eigenvalues are ordered in a non-decreasing fashion:

$$\lambda_i < \lambda_{i+1}, i \in \{0, ..., n-2\}. \tag{3.14}$$

Now the normal vector is simply the eigenvector e_0 with the smallest eigenvalue λ_0 of the Jacobian of the centroid $c(p_i)$ at position p_i.

$$n(p_i) = e_0. \tag{3.15}$$

Weight Functions for the Natural Normals Approach

In the stream processing system, three weight functions have been implemented for use with the natural normals.

Local Support This weight function was proposed by [Alexa and Adamson, 2009]. The size of the support is controlled by parameter h, as the weight goes to zero at exactly distance h.

$$\theta(r) = \frac{h}{r^2} - \frac{3}{h^3}r^2 + \frac{8}{h^2}r - \frac{6}{h}. \tag{3.16}$$

Power of -m Another weight function proposed by [Alexa and Adamson, 2009], this function smoothly goes to infinity as z approaches zero. Parameter m can be any positive number, but is usually set to 2.

$$\theta(r) = r^{-m} \tag{3.17}$$

Wendland's Weight A third and more complex function was proposed by Wendland [Wendland, 1995]. Similar to the local support weight function shown in Equation 3.16, a parameter h restricts the weight function to only include spatially close data points, here however by simply setting the weight to 0 once the distance gets too big.

$$\theta(r) = \begin{bmatrix} (1 - \frac{r}{h})^4(\frac{4r}{h} + 1) & 0 < r < h \\ 0 & r > h \end{bmatrix} \tag{3.18}$$

3.5.2.3 Triangle Normals

For the case that the input data set includes an additional stream containing triangular faces, the normal vectors can also be computed using the common approach used for triangular data sets.

Assume three vertices p_0, p_1 and p_2 that form a triangle. We now compute the vectors u and v corresponding to two edges of the triangle.

$$u = p_1 - p_0. \tag{3.19}$$
$$v = p_2 - p_0. \tag{3.20}$$

The face normal vector is now simply the cross product of those two edge vectors.

$$n_f = u \times v. \tag{3.21}$$

The surface normal vector n_i at point p_i can be obtained by summing up the face normal vectors $n_{f0}, n_{f1}, ..., n_{fn}$ of each triangle that contains p_i and normalizing the resulting vector.

3.5.3 Implementation

The two point-based normal estimation algorithms, covariance-based normals and natural normals, are implemented as stream operators and can be used interchangeably.

The covariance computation required for the covariance normal approach was extracted into an operator of its own. It is templatized using the high precision float type. This means that unless the user specifies differently, all covariance computation will be done in $double$ precision, and the resulting matrix is of type $double$.

The natural normals operator allows the choice of weight function and between using central or forward differences for filling the Jacobian as command line parameters.

Additional Operators

The normal operators write out a confidence value. An additional operator tests the confidence value and, if the confidence is too low, the replaces the low confidence normal with the average of the normals of all neighbors with high confidence values.

3.5.4 Results

High-quality normals are a requirement for high-quality point-based rendering. Whereas in polygonal rendering, the point position is crucial, and bad positions lead to visual artifacts, in point-based rendering the normal direction is much more important. An example of the visual effects of badly estimated normals is shown in Figure 3.8.

Synthetic Model Normal Quality

A model of a unit sphere was generated, with a size of 1000 points. Figure 3.9 displays the normal error on the sphere. Please note that the error scale ranged from 0.999 to 1.0, as both algorithms achieve a very high level of quality because of the smooth surface of the model.

Scanned Model Normal Quality

The quality of the estimated normals for non-synthetic models is measured with a comparison to vertex normals generated from triangle face normals from surface reconstructed mesh models. For this, the processing pipeline is run on each model two times: The first time with face information, to generate the vertex normals based on the corresponding face normals, and a second time to compute the estimated normal. The two normals are compared using the absolute value of the dot product, therefore larger numbers are better. We take the absolute value of the dot product as the orientation of the normals might skew the results otherwise (see Section 3.6). A value of 0.9 corresponds to an error of about 25°, while a value of 1.0 represents a perfect match. A visual comparison for the chinese dragon model is illustrated in Figure 3.10, with the color displaying the quality of the normal estimation. We can see a few patches of badly oriented normals around the eye and on some scales of the dragon.

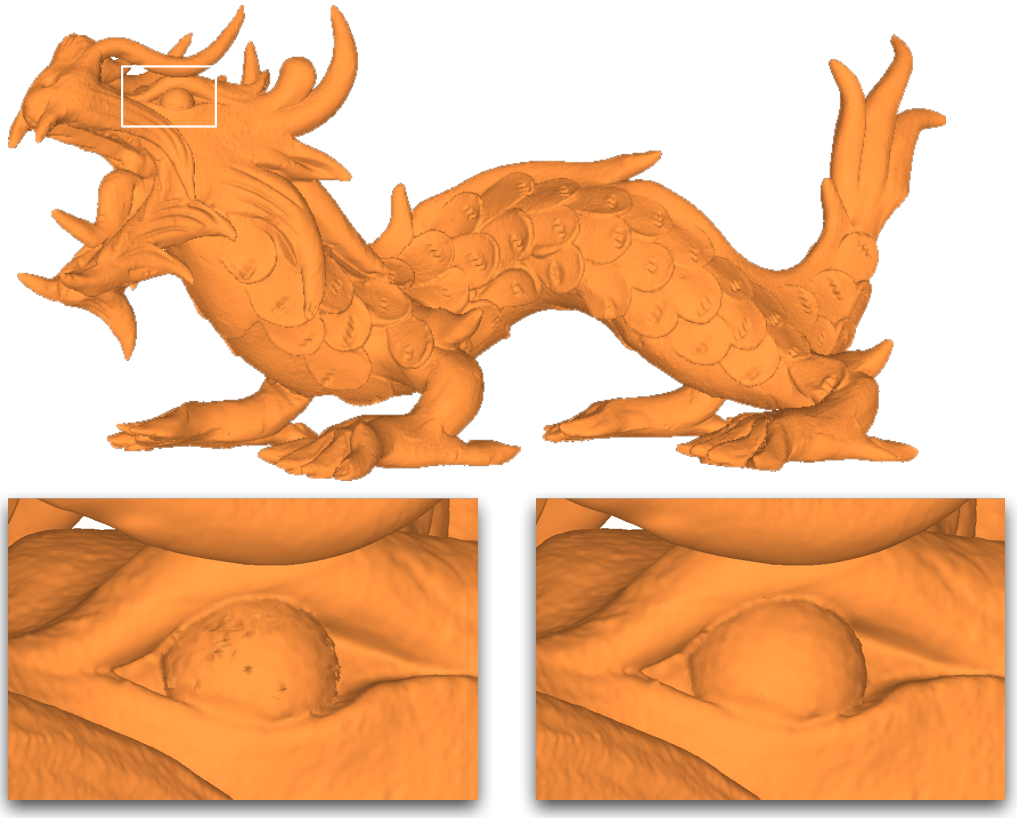


Figure 3.8: *The eye of the chinese dragon model was rendered twice, with two different sets of normals. The artifacts caused by bad normals are clearly visible in the left image. As a comparison, the right image shows the same model with identical settings, but a better set of normal vectors.*

The Effect of Neighborhood Size on Normal Quality

Various algorithms behave differently with a varying number of neighbors. Figures 3.11 and 3.12 show the behavior of the normal estimation to different neighborhood sizes. We can see that the quality of the normal estimation generally depends much more on the structure of the model than on the chosen neighborhood size. Compared to the natural normals operator, the covariance-based operator leads to more consistent results over different neighborhood sizes, however, the results of both operators with their respective optimal neighborhood size are close together. $k = 8$ seems to be an optimal size of the neighborhood for both operators irrespective of model. The results of both operators degrade with a too large neighborhood, however, this effect is much more pronounced when using the natural normals.

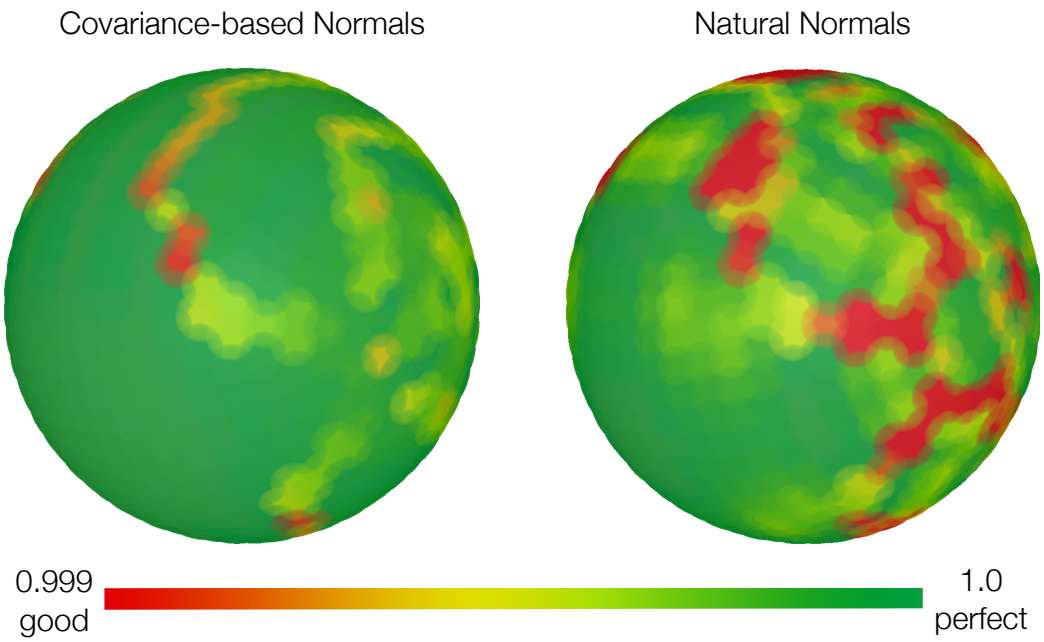


Figure 3.9: *A visual comparison of the normal quality on a generated sphere model with 1000 points. The normals for the left sphere were estimated using the covariance-normal operator, the normals for the sphere on the right with the natural normal operator, both with k = 8 neighbors. As a comparison, the generated (perfect) sphere normals were used. The error bar shows the colors used to represent the normal error.*

Algorithm and Model Structure

Comparing only the best results from the two operators as shown in Figure 3.13 we can see that there is no clear advantage for either one of the algorithms. The results depend more on the structure of the models, on factors as sampling density and the smoothness of the surface, and less on the algorithm used. Both algorithms show outliers. The covariance-based normals approach has problems with the david head model, while the natural normals approach cannot handle the statuette model very well, and the bunny model leads to bad results with both algorithms. It is likely that the low sampling density of the bunny model is the cause for this effect. We can again observe that the results achieved by the covariance-based normals operator are more consistent, in that even on the model with the largest advantage for the natural normals based operator, david head, the difference is far less pronounced than with the statuette model.

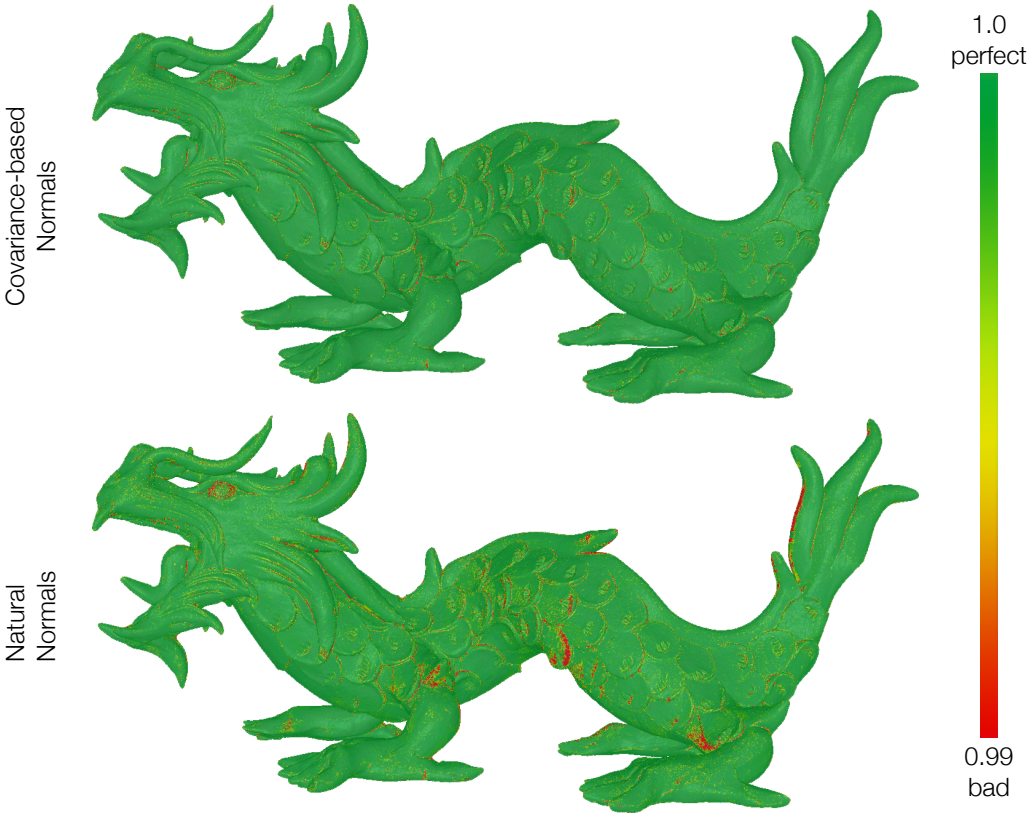


Figure 3.10: *A visual comparison of the normal quality on the dragon model. The normals for the upper dragon were generated using the covariance-normal operator, the normals for the lower dragon with the natural normal operator, both with $k = 8$ neighbors. The error bar shows the colors used to represent the normal error. 1.0 corresponds to a perfect match, while 0.9 means an error of 25° or more.*

Runtime Performance

Figure 3.14 shows the runtime performance of the covariance-based normal operator and the natural normal operator. We can see that computing covariance-based normals is consistently faster, taking only 80% of the time the natural normal operator uses, or 60% if we do not include the covariance computation, as the cost could be amortized if the covariance of the points can be reused in other operators.

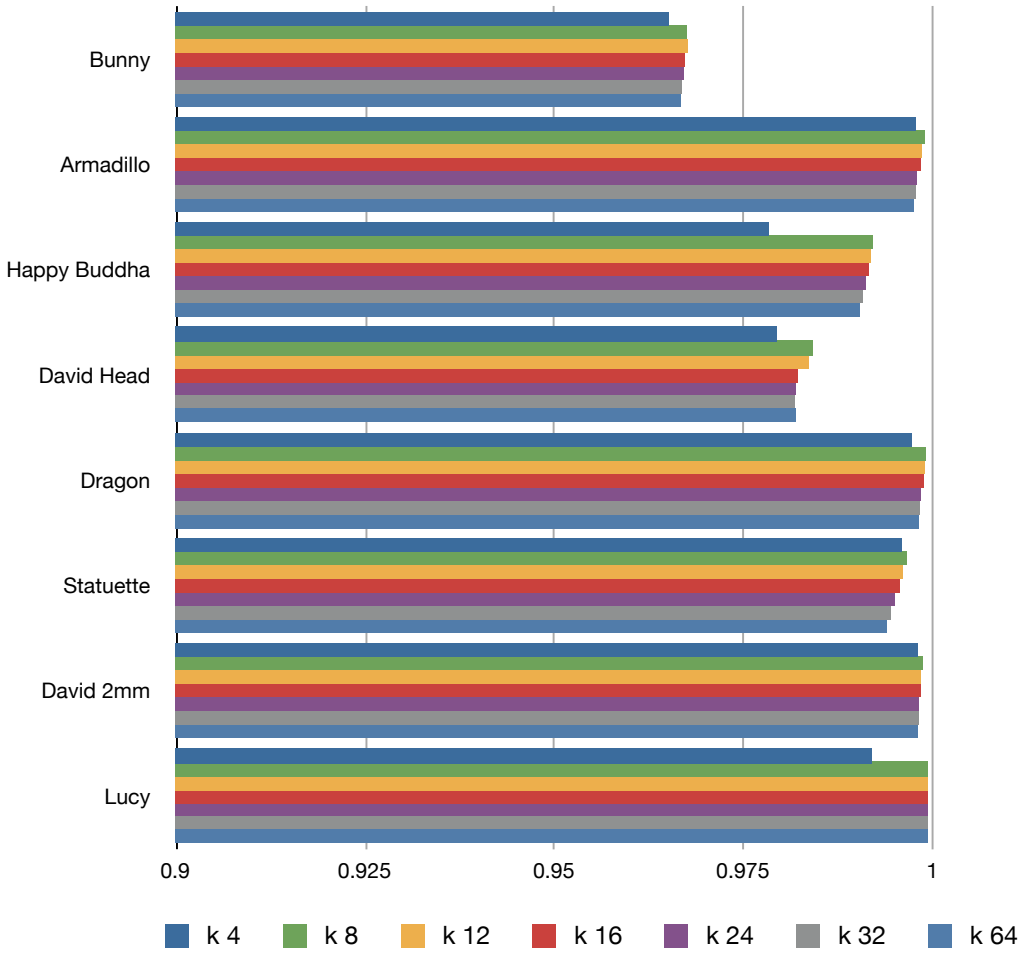


Figure 3.11: *Covariance-based Normals. A comparison of the normals estimated by the covariance-based normal operator with the normals computed from the triangles. Shown is the deviation between the normals as computed by the absolute value of the dot product, averaged over the data set. Large numbers are better. Each model was tested with different neighborhood sizes k as indicated. We can see that the quality of the neighborhood estimation is more dependent on the structure of the model than the chosen neighborhood size. Still, $k = 8$ seems to be an optimal number of neighbors for covariance-based normal estimation.*

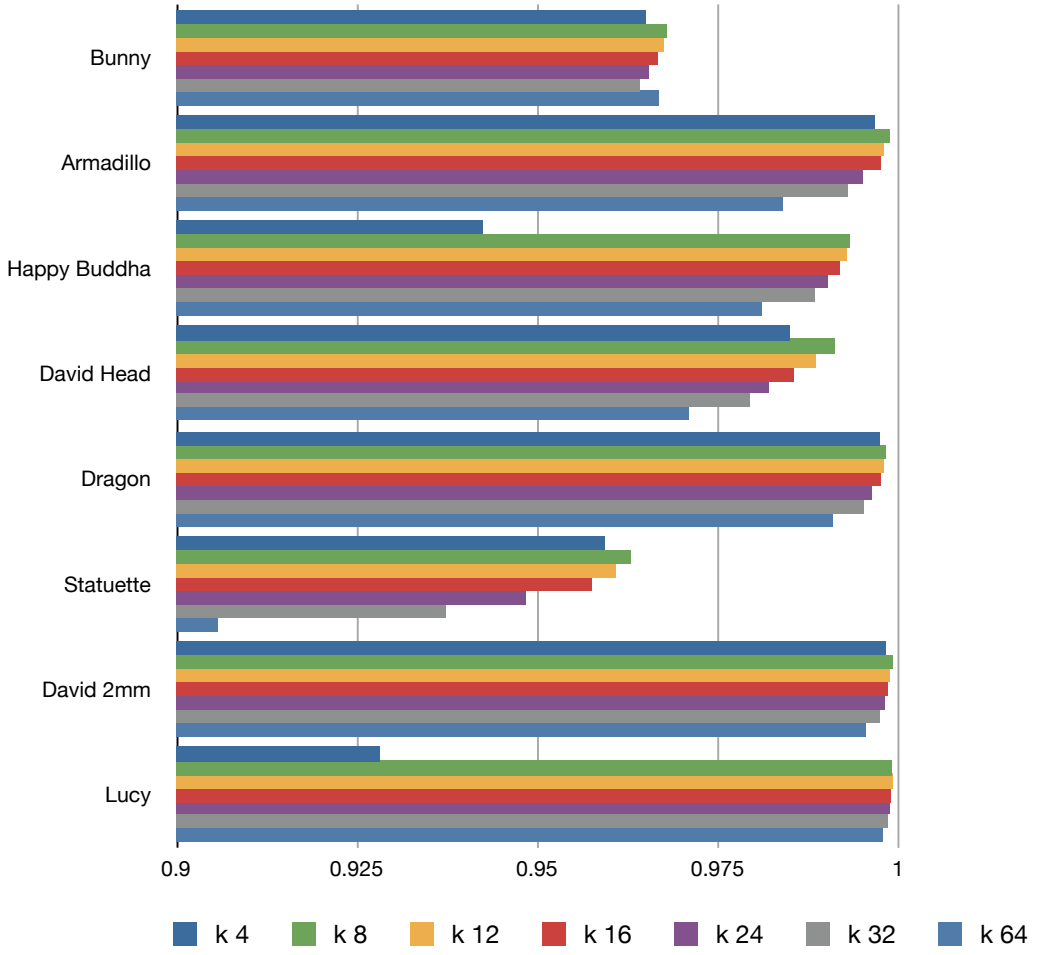


Figure 3.12: *Natural Normals. A comparison of the normals estimated by the naturals normals operator with the normals computed from the triangles. Shown is the deviation between the normals as computed by the absolute value of the dot product, averaged over the data set. Large numbers are better. Each model was tested with different neighborhood sizes as indicated. We can see that* $k = 8$ *leads to the best results, however, the quality of the estimated normals is much less consistent compared to the covariance-based approach, with multiple results below* 0.95.

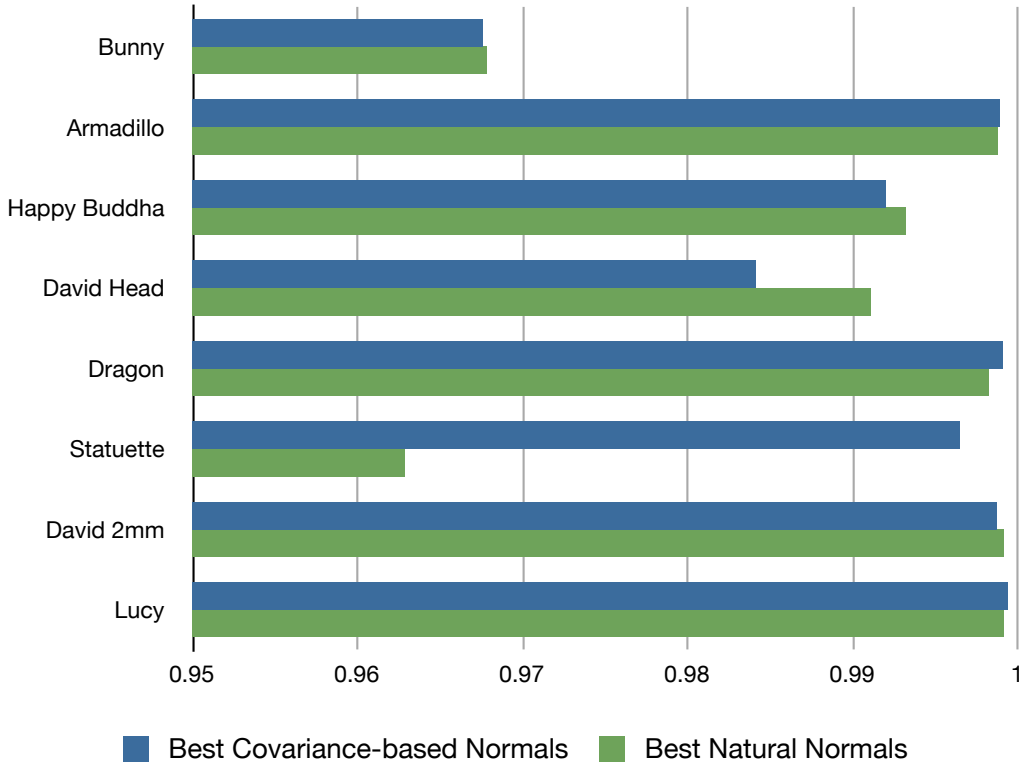


Figure 3.13: *A comparison of the normals estimated by the covariance-based normals operator with the naturals normals operator. Only the best result from the various neighborhood sizes is used. We can see that, in general, the results are close together. There is an outlier for the covariance-based normals on the david head model, and an outlier for the natural normals on the statuette model.*

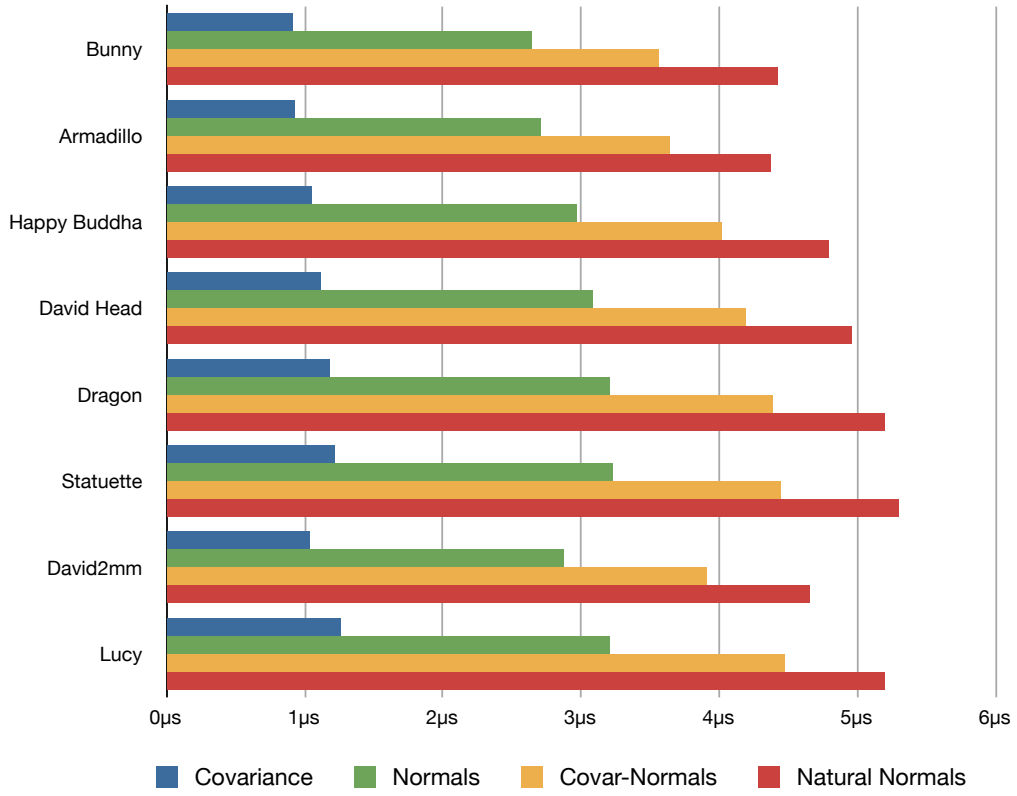


Figure 3.14: *A comparison of the runtime performance of the covariance-based normal operator and the natural normals operator. Shown is the time per point, that is, the total time for normal processing divided by the number of points. As covariance computation is a necessary step for the covariance-based normal operator, both the separate times as well as combined times are shown. We can see that computing a covariance normal requires only 80% of the time the natural normal operator takes, or 60% if we chose not to include the covariance computation, as it can be used in other operators as well.*

3.6 Normal Orientation

3.6.1 Background

Various approaches are known to efficiently estimate the surface normals of point sets, two of which are presented in Section 3.5.2.1 and 3.5.2.2. A major problem with these and many other approaches is that the estimated surface normal vectors are not consistently oriented. Some of the resulting surface normal vectors are inverted. A consistent normal orientation is a benefit to or even a requirement of many geometric algorithms. The most widely used approach for correcting the orientation of surface normals is normal propagation [Hoppe et al., 1992] based on minimum spanning trees. However, multiple passes are required in order to correctly apply Hoppe's algorithm in a streaming environment. We therefore chose a simpler technique instead.

3.6.2 Algorithm

For each point p_i, all its neighboring points $p_j \in \mathcal{N}_i$ are examined. We choose the subset of points $p_j \in \mathcal{N}_{i_{prev}}$ that are before the current point p_i in the stream and examine their normals n_j. The dot product between n_i and n_j is weighted using a weight function θ, and the weighted dot products are summed up. If f exceeds a threshold, the normal is flipped.

$$t = \sum_j \theta(n_i \cdot n_j). \tag{3.22}$$

The weight function θ is a simple distance function based on the euclidean distance d_i to the current neighbor and the distance d_{max} to the farthest neighbor.

$$\theta(x_i) = \frac{d_i}{d_{max}} * x_i. \tag{3.23}$$

3.6.3 Results

The two normal estimation algorithms presented in Section 3.5 lead to very different results when it comes to normal orientation. The natural normals approach leads to a many large patches, while with the covariance-based normals approach, more patches of smaller size appear, as can be seen in the left column of Figure 3.15.

The input to the normal orientation algorithm described above is therefore quite different depending on the normal estimation algorithm used, and this difference shows in the results. In Figures 3.15, 3.16 and 3.17, visual representations

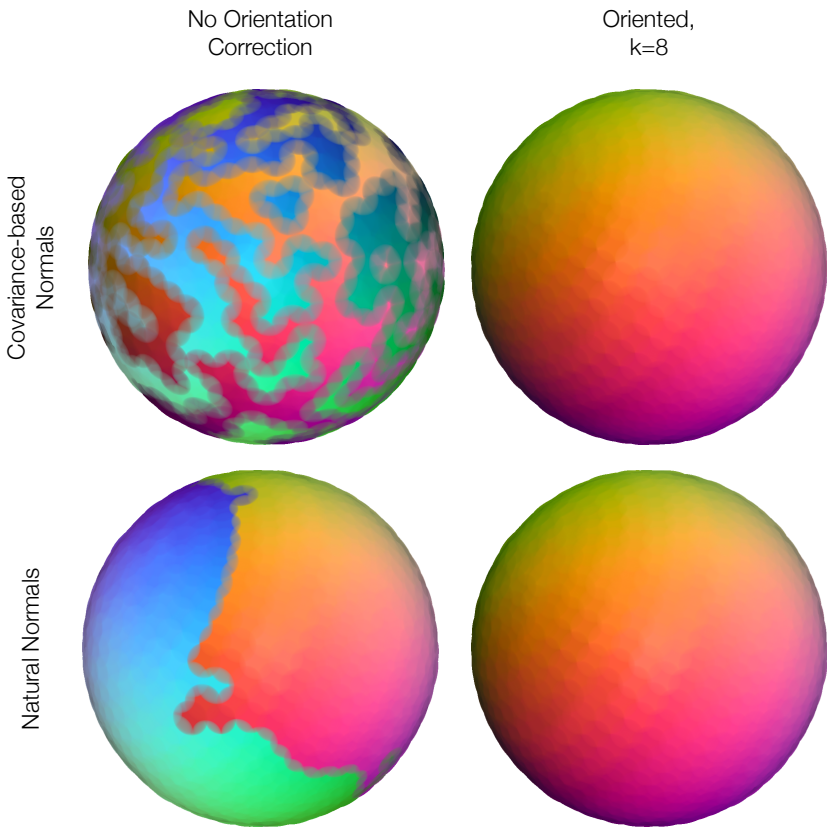


Figure 3.15: *The unoriented normals generated by the two normal estimation algorithms on the synthetic sphere model. The natural normals approach leads to a few large patches, while covariance-based normal estimation creates many smaller patches. After applying the algorithm, both spheres show correctly oriented normals.*

of the results of the normal orientation algorithm are shown. The models are rendered using colors according to the normal orientation so that patches of differently oriented normals are clearly visible. We can see in Figures 3.16 and 3.17 that the results vary greatly between the different models. The bunny model with natural normals and orientation with $k = 16$ neighbors has perfectly oriented normals, as with the armadillo model. Unfortunately, this algorithm doesn't perform that well on larger models such as the statuette model, where normal orientation leads to suboptimal results for both of the implemented normal estimation algorithms, with multiple patches of various sizes remaining after one pass of orientation, as shown in Figure 3.17. A more advanced normal orientation approach is necessary to guarantee consistently oriented normals for all models.

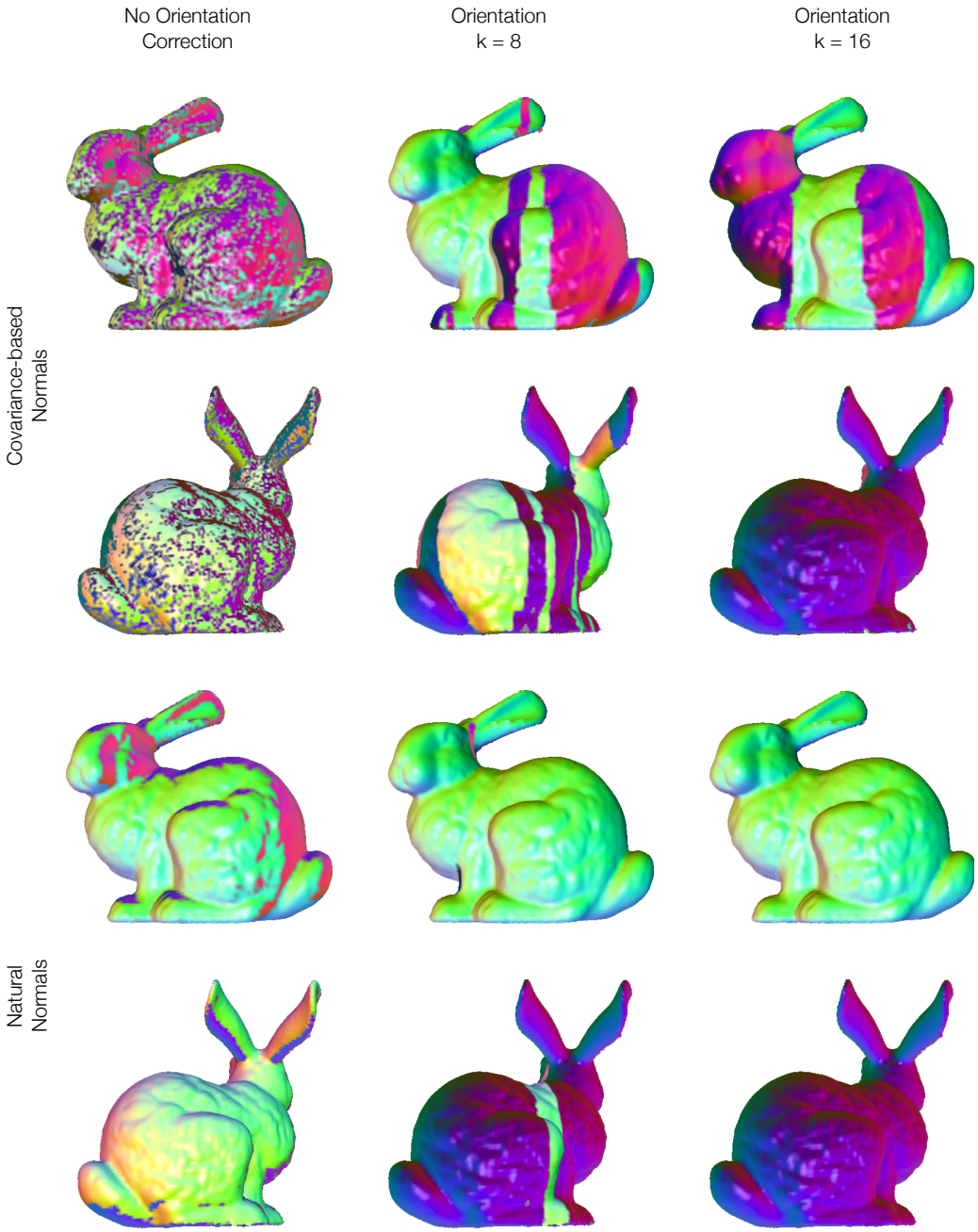


Figure 3.16: *A visual representation of normal orientation on the bunny model. Depending on the choice of normal estimation algorithm, the results range from acceptable to very good. With covariance-based normals and a neighborhood size of $k = 16$, only very few patches are left, and with natural normals and 16 neighbors, all normals are oriented correctly.*

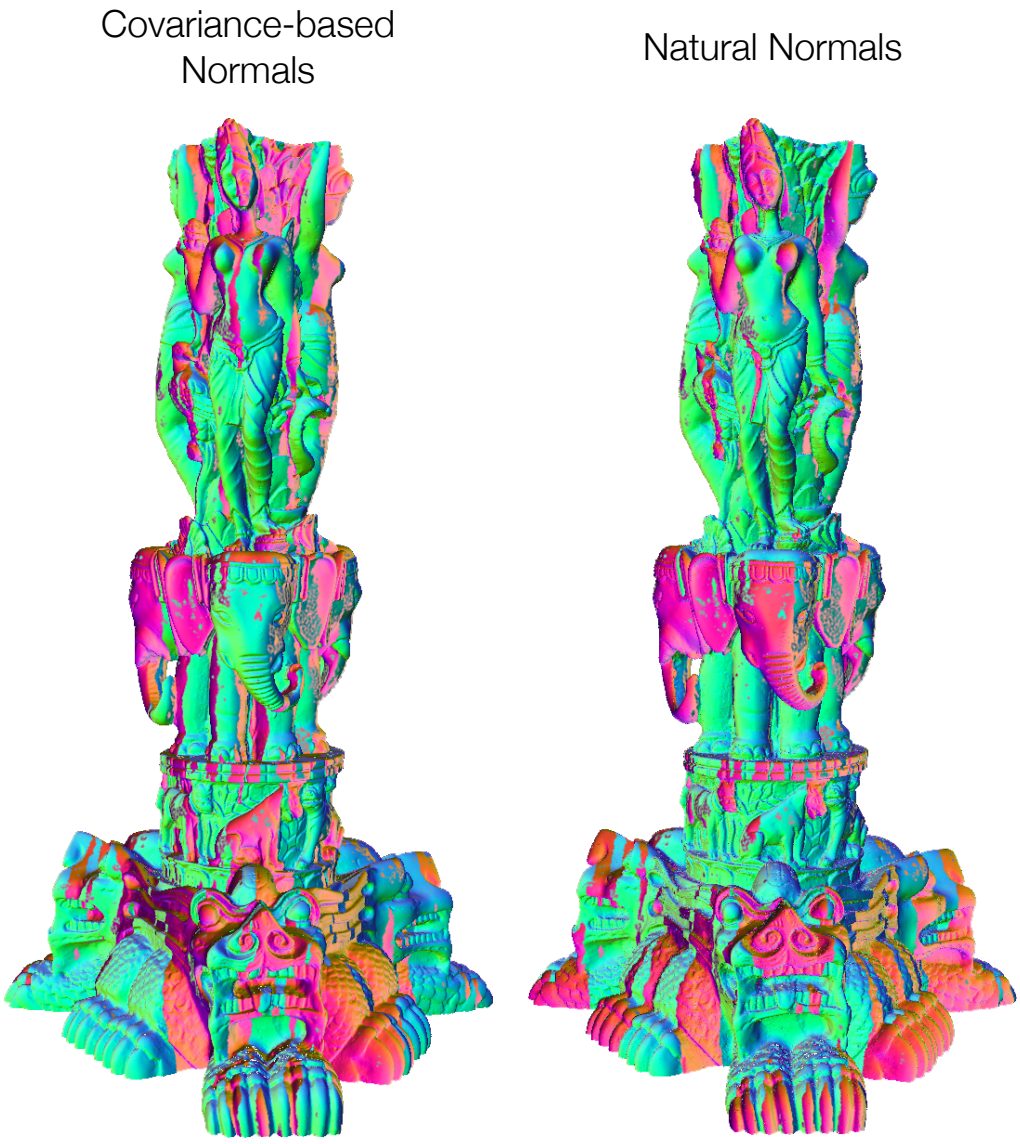


Figure 3.17: *An example where the orientation algorithm leads to suboptimal results. The normal orientation algorithm was run with $k = 16$ neighbors. We can see that in both cases, more than 10 patches of various sizes remain, wildly spread over the surface of the model.*

3.7 Sphere Fitting

3.7.1 Background

Point set surfaces have become an important tool for point processing as they allow approximation and interpolation of a point set. First introduced by [Alexa et al., 2003], [Amenta and Kil, 2004] proposed an approach based on using weighted centroids, which was improved on by [Adamson and Alexa, 2006] by simplifying the implicit surface definition.

The method we use in the stream processing system is the normal constrained algebraic sphere fitting proposed in [Guennebaud and Gross, 2007].

3.7.2 Algorithm

The algebraic sphere fitting is expressed as a minimization problem,

$$\mathbf{u}(\mathbf{x}) = \underset{u,u\neq 0}{\operatorname{argmin}} ||\mathbf{W}^{1/2}(\mathbf{x})\mathbf{D}\mathbf{u}||^2. \tag{3.24}$$

where $\mathbf{W}$ is the weight matrix

$$\mathbf{W}(\mathbf{x}) = \begin{bmatrix} w_0(\mathbf{x}) & & \\ & \ddots & \\ & & w_{n-1}(\mathbf{x}) \end{bmatrix}. \tag{3.25}$$

and $\mathbf{D}$ is the design matrix.

$$\mathbf{D}(\mathbf{x}) = \begin{bmatrix} 1 & \mathbf{p}_0^T & \mathbf{p}_0^T\mathbf{p}_0 \\ \vdots & \vdots & \vdots \\ 1 & \mathbf{p}_{n-1}^T & \mathbf{p}_{n-1}^T\mathbf{p}_{n-1} \end{bmatrix}. \tag{3.26}$$

A constraint is needed to avoid the trivial solution $u = 0$. Without normals, $Pratt's$ [Pratt, 1987] constraints can be used, but as the stream processing system includes multiple methods to compute or estimate normals, a second approach can be taken. Using the normals at point positions, the problem can be posed as a system of standard linear equations.

$$\mathbf{W}^{1/2}(\mathbf{x})\mathbf{D}\mathbf{u} = \mathbf{W}^{1/2}(\mathbf{x})\mathbf{b}. \tag{3.27}$$

with the weight matrix $\mathbf{W}$ defined as

$$\mathbf{W}(\mathbf{x}) = \begin{bmatrix} \ddots & & & & & & \\ & w_i(\mathbf{x}) & & & & & \\ & & \beta w_i \mathbf{x} & & & & \\ & & & \ddots & & & \\ & & & & \beta w_i \mathbf{x} & & \\ & & & & & \ddots & \end{bmatrix}. \tag{3.28}$$

and the design matrix $\mathbf{D}$:

$$\mathbf{D}(\mathbf{x}) = \begin{bmatrix} \vdots & \vdots & \vdots \\ 1 & \mathbf{p}_i^T & \mathbf{p}_i^T \mathbf{p}_i \\ 0 & \mathbf{e}_0^T & 2\mathbf{e}_0^T \mathbf{p}_i \\ \vdots & \vdots & \vdots \\ 0 & \mathbf{e}_{d-1}^T & 2\mathbf{e}_{d-1}^T \mathbf{p}_i \\ \vdots & \vdots & \vdots \end{bmatrix}. \tag{3.29}$$

As suggested by [Guennebaud and Gross, 2007], β is set to $10^6 h(\mathbf{x})^2$. $h(\mathbf{x})$ is a smooth function describing the local neighborhood size, defined as

$$h(\mathbf{x}) = \frac{\sum_i w_i(\mathbf{x}) h_i(\mathbf{x})}{w_i(\mathbf{x})}. \tag{3.30}$$

The pseudo-inverse method is used to solve the equation

$$\mathbf{u}(\mathbf{x}) = \mathbf{A}^{-1}(\mathbf{x}) \hat{\mathbf{b}}(\mathbf{x}) \tag{3.31}$$

with the weighted covariance matrix $\mathbf{A}$ and vector $\hat{\mathbf{b}}$ defined as follows.

$$\mathbf{A}(\mathbf{x}) = \mathbf{D}^T \mathbf{W}(\mathbf{x}) \mathbf{D} \tag{3.32}$$

$$\hat{\mathbf{b}}(\mathbf{x}) = \mathbf{D}^T \mathbf{W}(\mathbf{x}) \mathbf{b}. \tag{3.33}$$

3.7.3 Implementation

The sphere fitting is based on ExpeNG by [Guennebaud and Gross, 2007], implemented as a stream_operator.

3.7.4 Results

We use the sphere for estimating accurate Gaussian curvature, as described in Section 3.8. Results on the quality of the fitted sphere can be found in [Guennebaud and Gross, 2007].

3.8 Curvature Estimation

3.8.1 Background

Curvature estimation is an important area of research when dealing with surface models as the curvature contains information on many interesting geometric properties. Before going into the details of our approach, we provide a set of common definitions for dealing with curvature.

Curvature Vector The curvature vector k is defined for a curve $c(s)$ as the derivative of the unit tangent vector with respect to s.

Normal Curvature The normal curvature of a surface in a specific tangential direction is the reciprocal of the radius of the circle that provides the best approximation of a normal slice in that direction. The normal curvature vector k_n and the normal curvature κ_n can be defined with respect to any curvature vector k and the surface normal vector n_i.

$$k_n = (k \cdot n_i)n_i \tag{3.34}$$

$$\kappa_n = k \cdot n_i \tag{3.35}$$

Principal Curvatures The normal curvatures where the circle radii are the largest and smallest are called the principal curvatures and are denoted κ_1 or κ_{max} and κ_2 or κ_{min} , and the associated tangent vectors are the principal curvature directions e_1, e_2.

Gaussian Curvature The Gaussian curvature $\mathcal{K}$ at a certain point on the surface is defined as the product of the two principal curvatures:

$$\mathcal{K} = \kappa_1 \kappa_2 \tag{3.36}$$

Mean Curvature The mean curvature H is defined as the average of the two principal curvatures:

$$H = \frac{\kappa_1 + \kappa_2}{2} \tag{3.37}$$

3.8.2 Algorithm

Normal-Space Covariance

Curvature estimation in the stream processing system is based on the covariance of the normals of the points in the local neighborhood. Using the definition established in Section 3.5.2.1, the curvature at point p_i is based on the covariance Mn_i of the normals $n_{j0...n}$ of the points in the neighborhood $p_{j0...n} \in \mathcal{N}_i$, as shown in Equation 3.38.

$$Mn_i = |\mathcal{N}_i|^{-1} \cdot \sum_{p_j \in \mathcal{N}_i} (n_j - n_i)^T \cdot \theta(|p_j - p_i|). \tag{3.38}$$

An estimate of the curvature and the principal curvature directions can now be obtained by performing a singular value decomposition (SVD) on the normal space covariance Mn_i. The two principal curvature directions e_1 and e_2 are the eigenvectors with the largest eigenvalue and the cross product of that vector with the surface normal vector n_i. This method does not give us the value of the Gaussian curvature directly. However, we do get highly accurate principal curvature directions and the corresponding curvatures, the two curvature estimates κ_1', κ_2' with the right ratio, but unsigned and with an incorrect scale.

Curvature Scaling Using Fitted Spheres

The sphere fitting described in Section 3.7 can be used to scale the normal covariance curvature values κ_1' and κ_2'. The radius r_{sphere} of the sphere is an estimate of the mean curvature[4] H. Using κ_1', κ_2' and and r_{sphere}, we get a scale factor s. With this factor, we can compute the absolute values $|\kappa_1|$ and $|\kappa_2|$ of the principal curvature. If the sphere fit resulted in a plane or the curvature estimates are degenerate, we force $|\mathcal{K}|$ and therefore $\mathcal{K}$ to zero and do not perform any further steps.

$$s = \frac{H}{\frac{\kappa_1' + \kappa_2'}{2}} = \frac{r_{sphere}}{\frac{\kappa_1' + \kappa_2'}{2}}.$$
$$|\kappa_1| = s\kappa_1'.$$
$$|\kappa_2| = s\kappa_2'.$$

[4] see Section 3.37.

The Sign of the Gaussian Curvature

Using the scaled curvature estimates, we can compute the extent of the Gaussian curvature, i.e., the absolute value $|\mathcal{K}|$ of the Gaussian curvature $\mathcal{K}$.

$$|\mathcal{K}| = |\kappa_1| * |\kappa_2|. \tag{3.39}$$

In the presence of correctly a oriented normal that we assume to be outward pointing the sign of the Gaussian curvature can be determined using the dot product between the direction vector $v = center_{sphere} - p_i$ and the normal vector n_i.

$$d = v \cdot n_i \tag{3.40}$$

If the normal points into the same direction as v, the Gaussian curvature must be negative and we flip the sign on $|\mathcal{K}|$.

$$\mathcal{K} = \begin{cases} -|\mathcal{K}| & \text{if } d > 0 \\ |\mathcal{K}| & \text{if } d \leq 0 \end{cases} \tag{3.41}$$

We now have an accurate estimate for the Gaussian curvature $\mathcal{K}$.

Other Approaches

During our research on curvature estimation, we examined, implemented and tested different approaches for curvature estimation. Some of them did not perform as we had expected. In order to prevent others from repeating those experiments and potentially wasting their time, we will shortly present a number of things we discovered.

- *Curve sampling* [Agam and Tang, 2005] proposes an approach to estimate curvature based on curve sampling. A problem we encountered using this method were the requirements on sampling density. A curve is sampled using point p_i and two neighbors p_j and p_k, with an tangent-space angle α between $v_{ij} = p_i - p_j$ and $v_{ik} = p_i - p_k$. The method requires α to be very close to 0 or the computed curvature estimate will reflect the angle α much more than the actual curvature. With scanned models, a sampling dense and regular enough that we have pairs of neighbors p_{j0}, p_{j1} on a straight line with p_i is not realistic, and our test models did not give accurate curvature estimates using the method. A screenshot of our results with the method is shown in Figure 3.24 in the result section.

- *Paraboloid Fitting* The principal curvatures k_1, k_2 and the respective principal curvature directions e_1, e_2 can be expressed as a paraboloid. The coordinate system is formed by e_1, e_2, n_i, and the paraboloid parameters a and b can be transformed into the principal curvatures κ_1, κ_2. The sign of κ_1 and κ_2 determine the paraboloid type: if they match, the paraboloid is elliptic, otherwise hyperbolic. [Dai et al., 2007] proposed a method for least squares fitting of paraboloids, which we implemented. A problem we encountered was that their paraboloid fit is not restricted to only plane, hyperbolic and elliptic paraboloids, but can result in other fit types. Only a small subset of the fits actually resulted in one of the useful (for us) types, e.g. less than 20% on the dragon model.

3.8.3 Results

Visualization

Screenshots of rendered point models are used to visualize the Gaussian curvature. Before we present our results, we provide a quick explanation of the algorithm used to transform the Gaussian curvature $\mathcal{K}$ to a color value. We use a color texture, and transform $\mathcal{K}$ into a texture coordinate as shown in Equation (3.42).

$$f(x) = (log(|x|) + 10.0) * 0.025. \tag{3.42}$$

$$\text{tex\_coord} = \begin{cases} 0.5 - f(\mathcal{K}) & \text{if } \mathcal{K} < 0 \\ 0.5 & \text{if } K = 0 \\ 0.5 + f(\mathcal{K}) & \text{if } \mathcal{K} > 0 \end{cases} \tag{3.43}$$

Function f is based on the logarithm and transforms the absolute value of $\mathcal{K}$ to a value that can be used an offset into the color texture displayed in Figure 3.18.

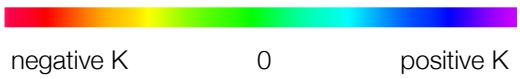


Figure 3.18: *The texture used for visualizing Gaussian curvature. Green corresponds to $K = 0$, deep sky blue to $K = 1$, etc.*

Gaussian Curvature on Synthetic Models

A first evaluation of our Gaussian curvature estimation method is performed by applying the algorithm to two synthetic models of which the correct Gaussian curvature is known. These models are a unit sphere with 100'000 points and a

cylinder with 125'000 points. The Gaussian curvature for an ideal sphere is a positive constant, in the case of the unit sphere $K = 1$. As the Gaussian curvature is the product of the two principal curvatures and a cylinder is planar in the direction of the minimum curvature, an ideal cylinder has a Gaussian curvature of $K = 0$.

The results of the test with synthetic models visualized in Figure 3.19, and displayed in Table 3.3. The estimation of Gaussian curvature resulted in correct results, with a small deviation from the actual values on the sphere. The cylinder curvature is exactly 0 due to either the sphere fit resulting in a plane, or because the curvature estimates k_1', k_2' indicated planarity.

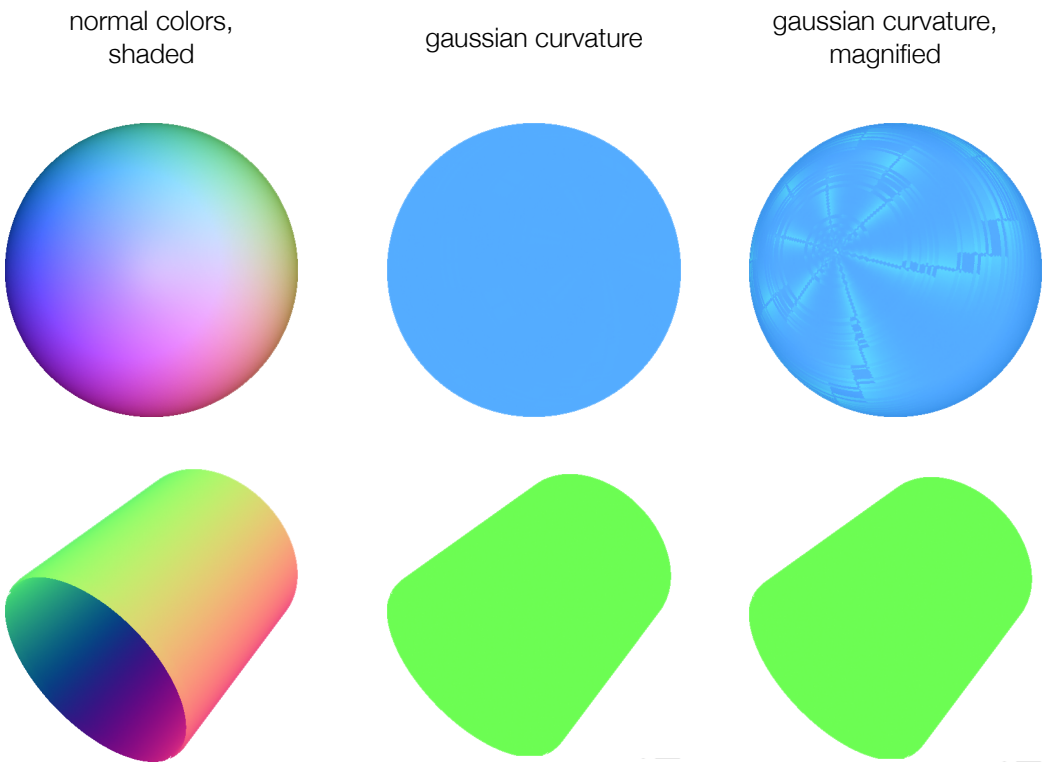


Figure 3.19: *Gaussian curvature displayed on the two synthetic models, a unit sphere and a cylinder. The left-most models are rendered with normal colors and shading, in the middle with Gaussian curvature and to the right with a strong magnification of the Gaussian curvature. While there are artifacts visible on the sphere on the right, they are only noticeable with strong magnification.*

Model	Min. K	Max. K	Average K	Avg. k1	Avg. k2
Sphere	0.91548	1.0034	0.986591	1.0912	0.908803
Cylinder	0	0	0	3.98367	0

Table 3.3: *Results of Gaussian curvature estimation on synthetic models, a sphere with 100'000 points and a cylinder with 125'000.*

Gaussian Curvature on Scanned Models

A next experiment was running the curvature estimation on actual scanned point models. A visualization of the Gaussian curvatures on the dragon model is shown in Figure 3.20. Two sections are highlighted and contrasted with a shaded rendering. We can see that the curvature estimation performs well even with the difficult Dragon model with its highly detailed surface with many small bumps and dents.

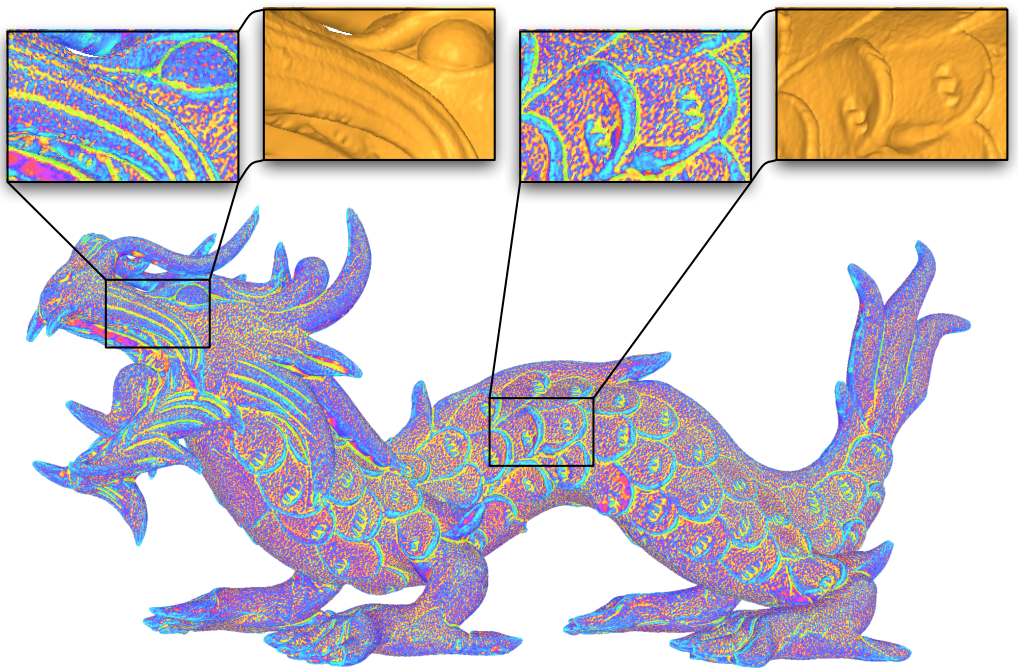


Figure 3.20: *Gaussian curvature displayed on the Dragon model. Note how the curvature estimation captures the highly detailed structure of the surface with its many dents and bumps. Two area are shown in greater detail and contrasted with a shaded rendering.*

In Figure 3.21, the Gaussian curvature is shown on the Statuette model. This model has sections with a much smoother surface than the dragon model, and the curvature shows many soft transitions.

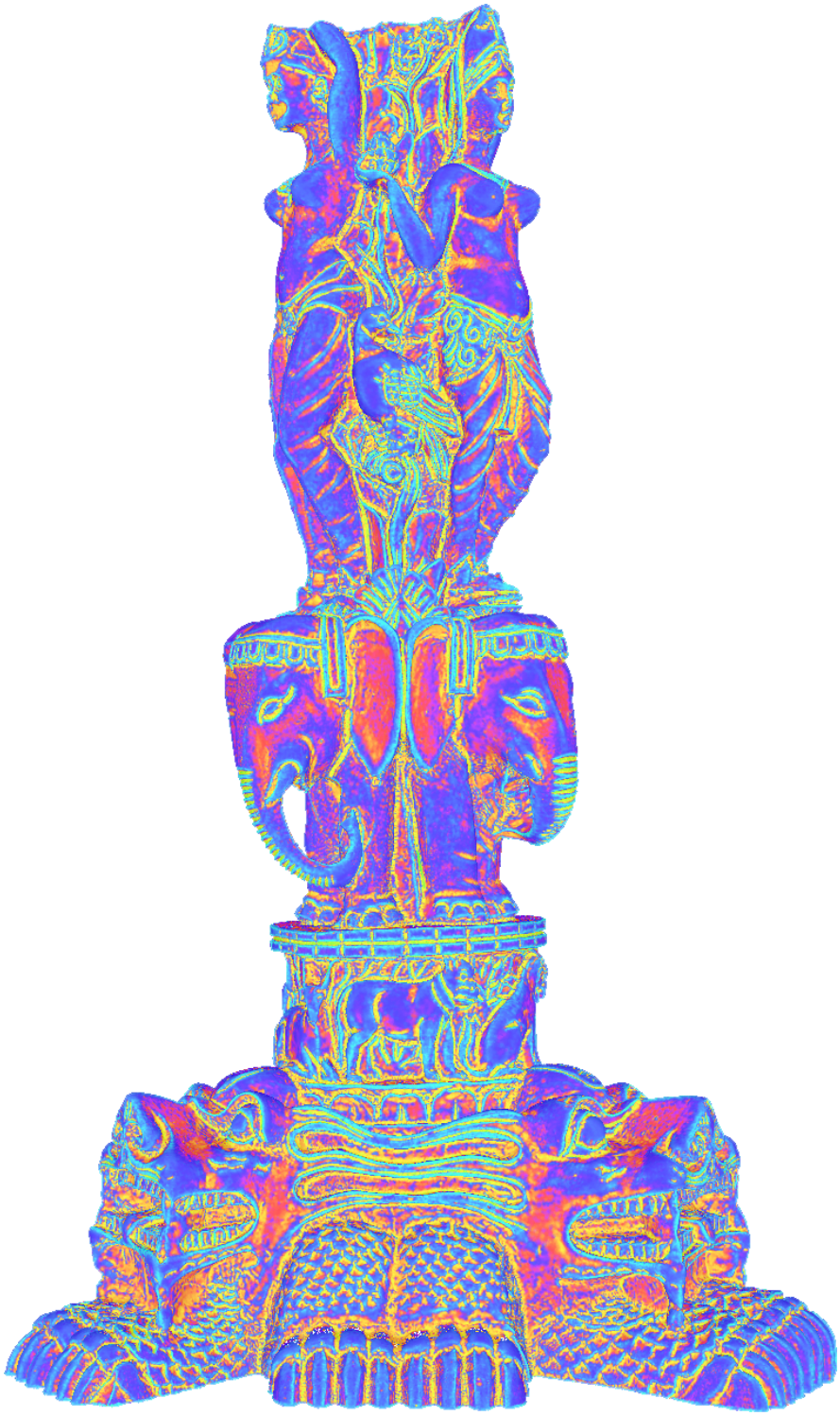


Figure 3.21: *Gaussian Curvature on the Statuette model.*

As explained in Section 3.5, the normal estimation implemented in the stream processing system cannot guarantee consistently oriented normals. This is a problem, as our Gaussian curvature estimation depends on consistently oriented normals. We therefore used normals computed from faces[5] for above experiments.

In the absence of consistent normals, the sign of the Gaussian curvature cannot be determined reliably, and we therefore only estimate $|\mathcal{K}|$. The dragon model with unsigned curvature is shown in Figure 3.22 for comparison.

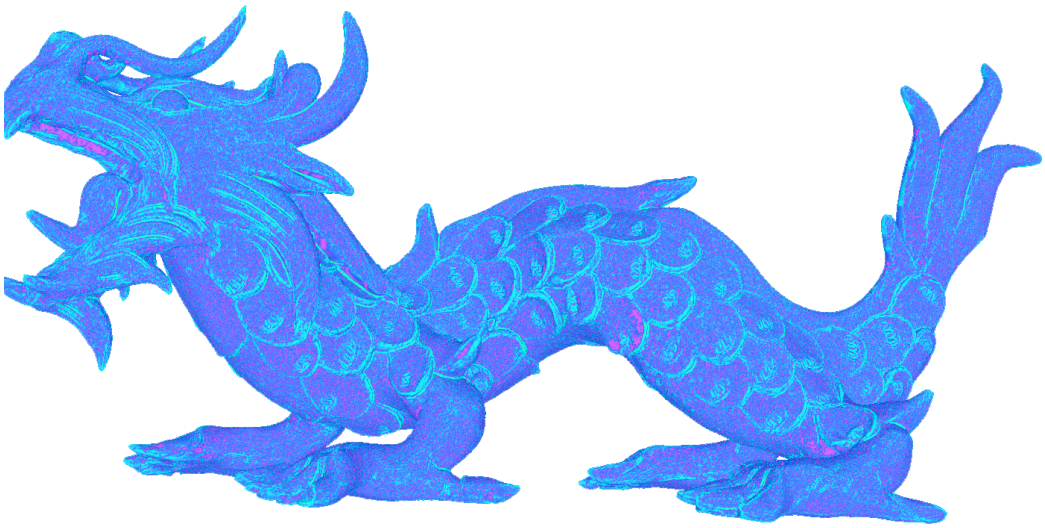


Figure 3.22: *Unsigned Gaussian curvature displayed on the dragon model.*

The normal orientation approach presented in Section 3.6 results in multiple patches of consistently oriented normals. Each patch with inverted normals also inverts the Gaussian curvatures of the points belonging to that patch. For an example of this, the curvature estimation was performed with normals computed by the covariance-based normal operator and partially oriented with the normal orientation operator. Figure 3.23 presents a visualization of the this inversion effect, contrasted with the correct result and a visualization of the normal orientation.

As mentioned in Section 3.8.2, other methods of computing curvature were examined as well. Figure 3.24 presents visualization of the results we achieved using the curvature sampling approach [Agam and Tang, 2005].

To summarize, we propose a method to accurately estimate Gaussian curvature based on normal-space covariance and a sphere fit. However, the method requires consistently oriented normals, which is something that the stream processing system is not yet able to guarantee.

[5] See Section 3.5.2.3.

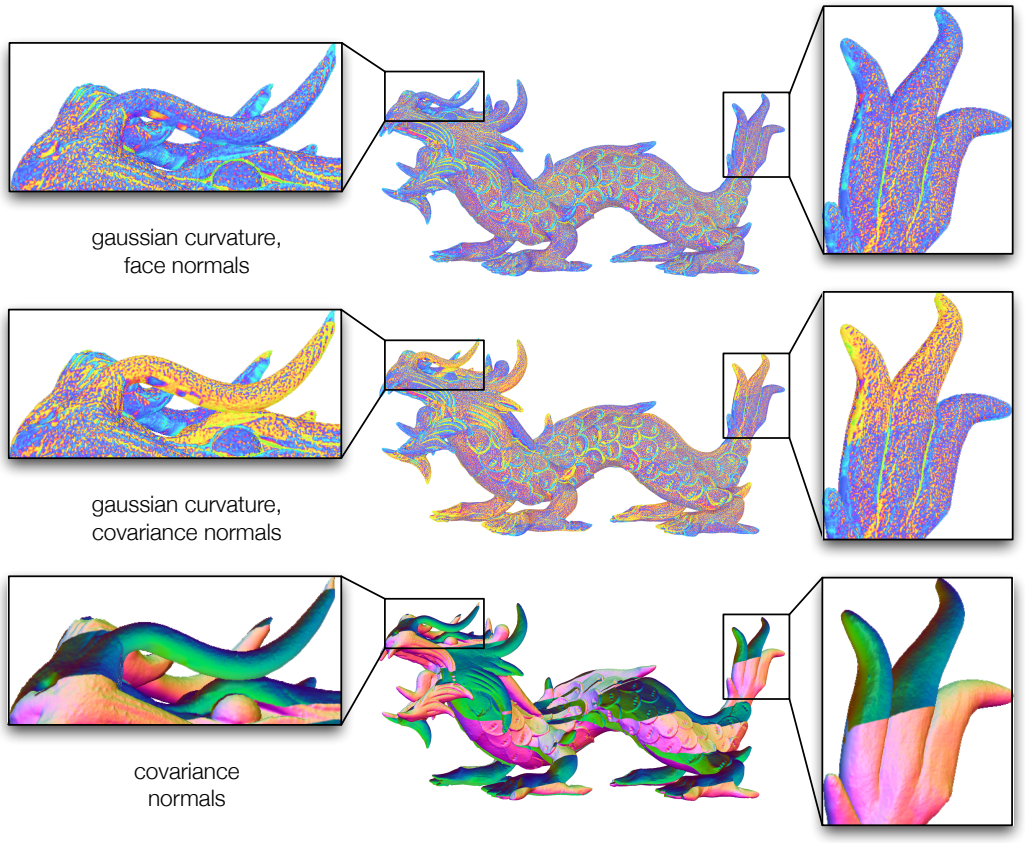


Figure 3.23: *The Gaussian curvature estimation requires consistently oriented normals in order to estimate not only the extent, but also the sign of the Gaussian curvature. Note how the curvature estimate is inverted in patches of inconsistently oriented normals.*

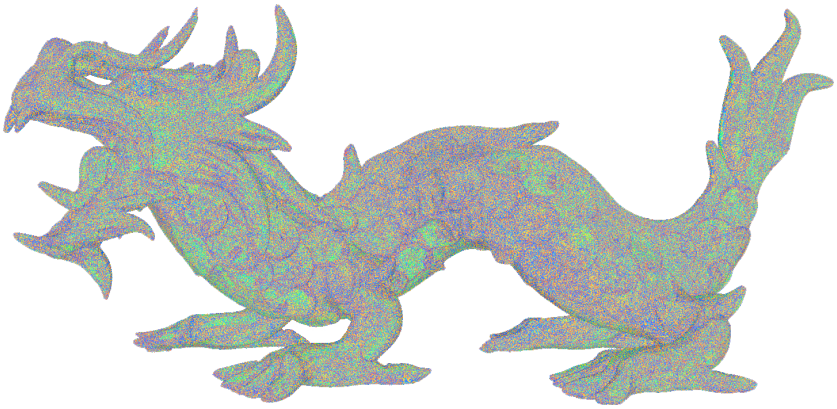


Figure 3.24: *Curvature estimation on the dragon model using the curve sampling approach presented in Section 3.8.2, based on [Agam and Tang, 2005]. As the angles between pairs of neighbors are large, the curvature estimates are not accurate.*

3.9 Ellipse Splat Estimation

3.9.1 Background

The traditional approach to render models is the use of polygonal meshes, most commonly consisting of triangles or quadrilaterals. With the availability of powerful GPUs, alternative techniques have been developed. High quality point-based rendering methods [Zwicker et al., 2004; Pfister et al., 2000; Rusinkiewicz and Levoy, 2000] use point splats to display a surface, with an additional blending step added to further improve the quality of the rendered image. An extension that includes the capability for transparency for point-based rendering was presented in [Zhang and Pajarola, 2006a], and [Hübner et al., 2006] proposed a method for single-pass multi-view point-splatting for stereoscopic devices. More information and an overview of different high-quality point-based rendering techniques on the GPU is presented in [Gross and Pfister, 2007a; Botsch et al., 2005; Sainz and Pajarola, 2004].

The point splats as used in point-based rendering methods can be either discs or ellipses aligned to the surface tangent. The rendering using discs performs better in terms of computational cost because of reduced complexity, but using ellipses allows to approximate a surface more closely, leading to a higher quality of the resulting image. Note that the second approach uses splats that are ellipses in objects space, not object-space discs that are rasterized to screen-space ellipses such as the first method.

In both cases additional point attributes are needed, as the basic point attribute is simply the point position p_i.

The rendering of disc-shaped splats requires the normal vector n_i to orient the splat and the point radius r_i to limit the splat extent. In the stream processing system, these are computed using the techniques presented in Section 3.4 and 3.5.

To render elliptical splats, in addition to the surface normal, the direction and length of the major splat axis u_i and the aspect ratio a_i between the major and minor axes are required. These can be computed using the results of the curvature estimation presented in 3.8.

3.9.2 Algorithm

Covariance analysis is commonly used to estimate the parameters of elliptic point splats [Pauly et al., 2002b; Pajarola, 2003; Pajarola et al., 2004]. With the principal curvature directions estimated in the curvature operator (see 3.8), we can use those directions as splat axes and only have to scale the splat to the size of the neighborhood. The major axis corresponds to e_2, the principal curvature direction

with the minimal curvature k_2, the minor axis to e_1 with curvature k_1. This way, the longer axis of the ellipse aligns with the direction of least curvature to better approximate the surface.

3.9.3 Results

To present the results of the splat estimation, we use a rendering algorithm similar to [Botsch et al., 2005] to display the model using ellipse splats. Figure 3.25 shows a rendered image of the Stanford Bunny model with both point- and ellipse-splats. We can see that the ellipse splats capture the shape in greater detail, and the ellipses are oriented in minimal curvature direction thanks to the curvature estimation described in Section 3.8.

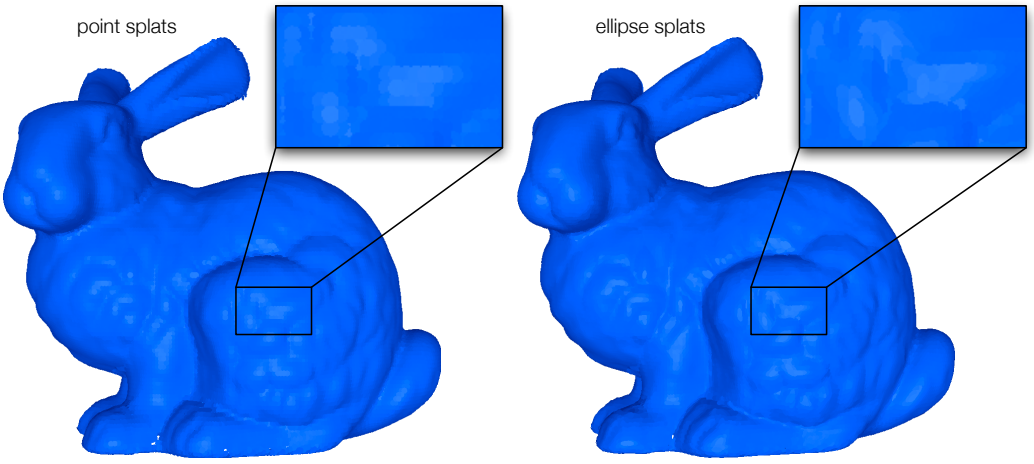


Figure 3.25: *An example of rendering with point splats and with ellipse splats. Note how the ellipse splats capture the shape in greater detail.*

A problem that often occurs when rendering scanned models with point splats are holes in the surface, as seen in Figure 3.25. If the sampling density is low or the points are distributed unevenly, such artifacts may occur. Increasing the point radius will cover the surface, but at the cost of a loss of detail. Ellipse splats can alleviate this problem as they cover the surface more accurately, especially when employed together with blending in order to create higher-quality images. A comparison between rendered images of a model with point radii too small to cover the surface, point radii have been increased at the cost of a loss of detail and a higher quality version using ellipse splats is shown in Figure 3.26.

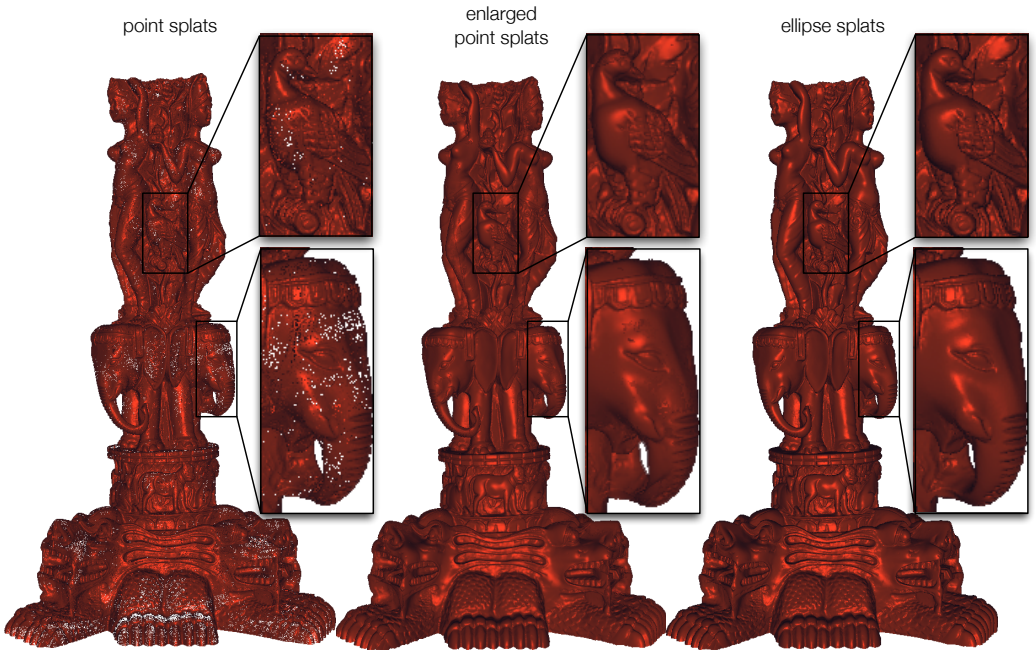


Figure 3.26: *A comparison between a model with point discs with a radius that doesn't cover the whole surface, a version where the radii have been increased at the cost of a loss of detail and a higher quality version using ellipse splats that more accurately cover the shape.*

3.10 Reeb Graph

3.10.1 Background

A reeb graph describes the connectivity of level sets or contours for a smooth function defined on a manifold, as a function value or iso-value is varied [Reeb, 1946]. It represents an abstraction of the topology, and is useful for a wide variety of applications. Computing a reeb graph is performed by contracting connected components of level sets to a point.

We define a specialization of the reeb graph for discrete point sets, the discrete reeb graph [Anwar, 2009], and an extension of the discrete reeb graph that embeds the reeb graph within a model.

The work discussed in this section is based on earlier work by [Anwar, 2009], which in turn is based on a previous version of the stream processing system originally presented in [Bösch and Pajarola, 2009].

Related Work

Reeb graphs are used as an abstraction of the topology of geometric models and have been used for finding topologically similar geometric models [Hilaga et al., 2001; Funkhouser et al., 2005; Steiner and Fischer, 2001] and for topological simplification [Guskov and Wood, 2001; Wood et al., 2004]. Other uses are computer aided geometric design [Shinagawa et al., 1995] and level set computation [Carr et al., 2004]. [Bajaj et al., 1997; Weber et al., 2007] used reeb graphs for specifying transfer functions for volume rendering. Another use for reeb graphs are operations on surfaces such as compression, reconstruction, embedding and parametrization [Shinagawa et al., 1991; Takahashi et al., 1997; Biasotti et al., 2000; Attene et al., 2001; Biasotti and Ricerche, 2001; Hétroy and Attali, 2003; Zhang et al., 2005].

Algorithms to compute reeb graphs typically operate on meshes, and not on point sets. An early approach proposed by [Shinagawa and Kunii, 1991] for triangle meshes offered a runtime complexity of $O(n^2)$. An improved algorithm presented by [Cole-McLaughlin et al., 2004] reduced that to O(nlog(n)).

For models without loops, contour trees are equivalent to reeb graphs. [Carr et al., 2000] proposed an algorithm to compute contour trees in O(nlog(n)) time. [Hilaga et al., 2001] compute an approximation of the reeb graphs.

These algorithms assume that the whole model fits into memory. [Pascucci et al., 2007] presented a streaming approach that operates on triangulated meshes.

3.10.2 Algorithm

Discrete Reeb Graph

We assume a graph G(V,E) consisting of a set of vertices V and edges E, and a discrete function f.

$$G = (V, E) \tag{3.44}$$
$$V = \{v_i\} \tag{3.45}$$
$$E = \{(v_i, v_j) | v_j \in \mathcal{N}_i\} \tag{3.46}$$

$$f :\rightarrow \mathcal{R} \tag{3.47}$$

Then we define the discrete level set as

$$f^{-1}(c) = \{(v_i, v_j) \in E | f(v_i) \leq c \leq f(v_j)\} \tag{3.48}$$

Additionally, we define the neighborhood $\mathcal{N}$ for a given iso-value by

$$\mathcal{N}_{e,c} = \{e\prime | \exists v : \begin{smallmatrix} v \in e \wedge e \in f^{-1}(c) \\ v \in e\prime \wedge e\prime \in f^{-1}(c) \end{smallmatrix}\} \tag{3.49}$$

Using these definitions, we arrive at a neighborhood graph.

$$NG_c(f^{-1}(c), \bigcup_{e \in f^{-1}(c)} N(e, c)) \tag{3.50}$$

This neighbor graph NG_c defines the connectivity of the contour at c, and a discrete reeb graph can be defined by tracking the connected components of NG_c over the values of c [Anwar, 2009].

For the stream processing system, the function value used is the vertex coordinate of each point on the streaming axis, which as shown in section 2.4.1 is always the z axis. Therefore, the reeb graph function value used in the stream processing system is simply the z component of the position vector of a vertex.

An in-depth discussion of the relationship between a reeb graph and a discrete reeb graph of a model can be found in [Anwar, 2009].

Computation of the Discrete Reeb Graph

The construction of a reeb graph as a set of vertices and edges as defined in equation 3.44 of a stream_process model is performed as a multi-step process.

First, the neighborhood graph of a data set is transformed element by element into an augmented reeb graph, such that each stream_data element is a vertex of the reeb graph, and each nearest-neighbor element is connected by an edge in the reeb graph. The position vector of each element within the reeb graph is copied into the reeb vertex, as most stream_data elements will be streamed out quickly while its associated reeb vertices potentially stay in memory much longer.

The resulting graph is called augmented reeb graph, as it is of a much higher degree initially and contains many superfluous edges and vertices that do not represent an actual change in connectivity.

We define a link as a set of edges in the augmented reeb graph that span a certain function value. The function value spanned by each edge is defined by the vertices it is incident to, and so each link also includes the vertices associated with its edges. Edges that are incident on a certain vertex and span a certain range of the function value are connected to this vertex, and therefore always remain in the same component of the discrete reeb graph. By varying the function value, we can use the links to trace paths taken by edges connected to a certain vertex, and associate these paths with the vertex.

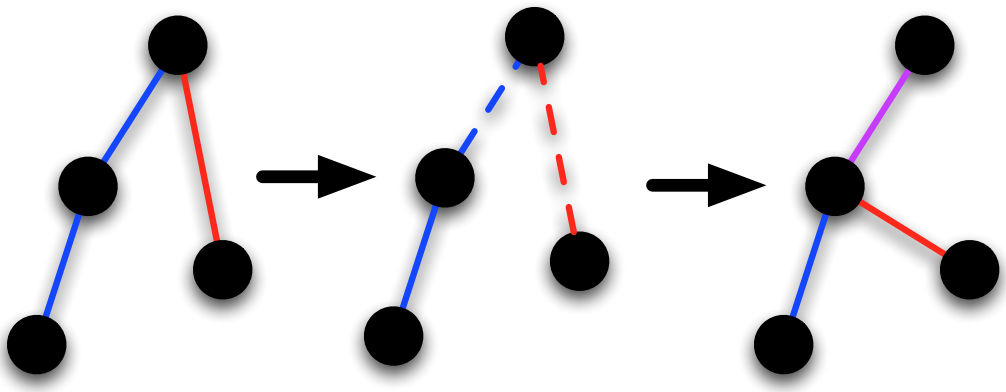


Figure 3.27: *Conceptual representation of the edge merging process in the reeb graph.*

After the insertion of a new vertex into the discrete reeb graph, a culling step is performed to remove the superfluous vertices and edges using gluing and splitting operations. Assume two paths between vertices v_i and v_j overlap. We then determine the link in v_j that spans the function value of v_i. If v_i and v_j do not lie on the same path, we split links so that v_i and v_j lie on the same path. If one or both of the new paths do not span the whole value range of the edge, a new link is created to the closest vertex on the existing path.

The next step is to glue the paths of those vertices together in a top-down fashion over the spanned range. If a loop is detected, which means that there are two different paths between v_i and one of the vertices in the range, one of the paths is removed. If the paths have different lengths, we merge the additional vertices into the remaining path. This merging and gluing is continued until we reach v_j. The path gluing process is visualized in Figure 3.28, and the edge merging in Figure 3.27.

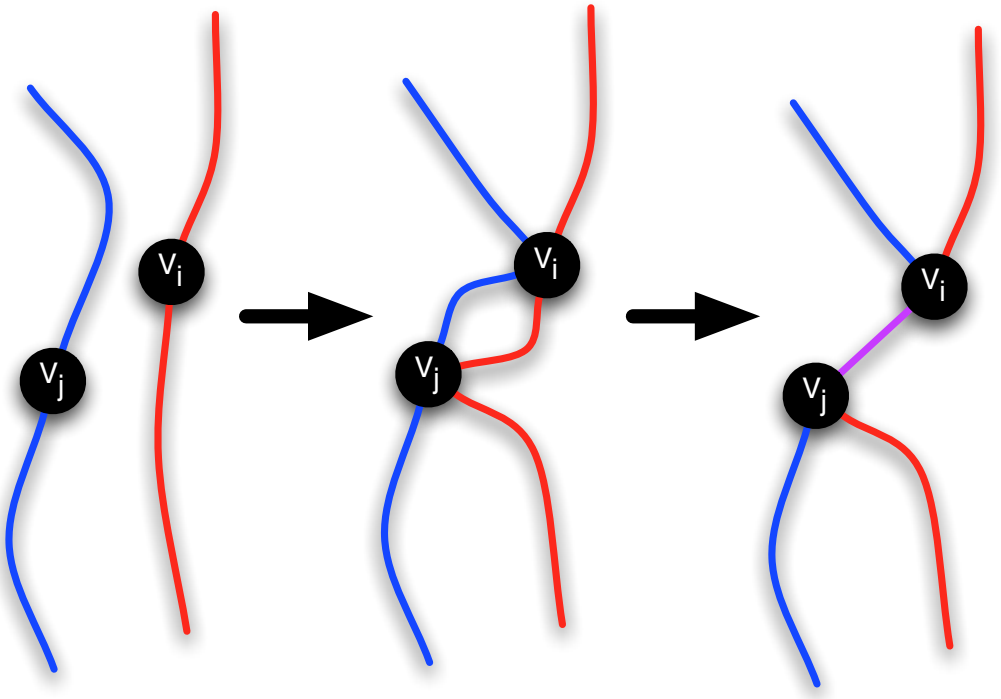


Figure 3.28: *Conceptual representation of the path gluing using for removing superfluous elements in the reeb graph.*

Embedded Discrete Reeb Graph

A discrete reeb graph consists of only nodes that correspond directly to the position of a point in the data set. With continued pruning of the graph, the original position loses some of its significance, as the node now represents the original position and also the nodes that were removed from the graph in its neighborhood. An embedding of the reeb graph in the model tries to solve that problem by moving the position of the node in reaction to the purging of directly connected nodes. In this work, three embedding algorithms were implemented.

Embedding Algorithms

The three algorithms used for embedding the reeb nodes in the point data set are described below. The algorithms share most of the logic in that whenever a node is removed from the reeb graph, all neighboring nodes[6] are updated with the position of the removed node.

Naive Embedding The naive embedding algorithm directly updates the position of each neighboring node when a node is purged. This is done by computing the translation vector between the two node positions and moving the surviving node half-way towards the purged node.

2D Embedding

The 2D embedding algorithm accumulates the positions of the purged nodes within the remaining nodes. However, it ignores the position component in the streaming direction. So for a coordinate system with three axis x, y, z, and z being the streaming direction, only the x and y components are considered. The two non-streaming-axis components are added up, and count the number of position updates. After processing is stopped and the reeb graph is complete, we divide the embedded positions by the counter in order to get the correct embedding positions.

Weighted 2D Embedding

The weighted 2D embedding algorithm is an enhancement of the 2D embedding in that each reeb neighbor node's contribution to the embedding node is weighted. Different weights have been implemented, based on the distance in streaming direction.

$$w = \theta(|z_i - z_{nb}|). \tag{3.51}$$

As in the 2D embedding algorithm, the component of the position in streaming direction is not updated with the neighbor's information.

Filtering A filter can be optionally applied to the computed reeb graph that removes all nodes with a degree of 1, that is, all hanging leaf nodes in the graph that are connected to only a single neighboring point.

[6]Neighboring nodes in this context are nodes that share an edge in the reeb graph. This reeb neighborhood is does not directly correspond to the k-nearest-neighborhood definition presented in Section 3.3. This means that reeb nodes may be neighbors in the reeb graph without being part of the corresponding point's neighborhood set $\mathcal{N}$.

3.10.3 Implementation

The reeb graph and embedded reeb graph algorithms have been implemented as templateized classes than can be used as either stream_operator[7] or chain_operator.

Upon insertion of a new stream_data element, a node in the reeb graph is created, its vertex position is copied into the reeb node, and edges to the nodes of its neighboring points are generated. Then, the pruning algorithm described above is performed. The remaining nodes stay in memory until processing is completed. This is not a problem even for large data sets, as the reeb graph comprises only a tiny subset of the whole model, as shown in Section 3.10.4.

The reeb graph is written out as an ascii or binary .ply file consisting of the vertices, the embedded vertex positions and the edges of the graph. Optionally, the .tlp file format used for the tulip graph visualization application [LaBRI, 2011] is supported. As the output writer classes are separate from the actual reeb stream_operator, additional output format can be implemented non-intrusively.

3.10.4 Results

As examples for the reeb graph computation, we present embedded reeb graphs created by the reeb operator with various parameters.

Embedding the Discrete Reeb Graph

In Figure 3.29, two reeb graphs computed for the armadillo model are shown, the discrete reeb graph and an embedded discrete reeb graph. As each node of the non-embedded discrete reeb graph must directly correspond to a vertex of the input point set, two neighboring nodes of the non-embedded graph are often on different sides of the model, and the graph does not capture the shape of the model well. In the embedded version, the graph more closely resembles the model structure. The graphs displayed in the figure were generated with $k = 128$ neighbors and no filtering.

Neighborhood Size

The computation of the reeb graph is dependent on the size k of the point neighborhood[8], as only links to the points neighbors are created upon insertion in the graph. A too small neighborhood will eventually limit simplification of the graph, while a large neighborhood increases the runtime of the algorithm significantly.

[7] stream_operator and chain_operator are the two different types of operators in the stream processing system, see Section 2.2.2 for more information.

[8] More information on a points' neighborhood and neighborhood computation can be found in Section 3.3.

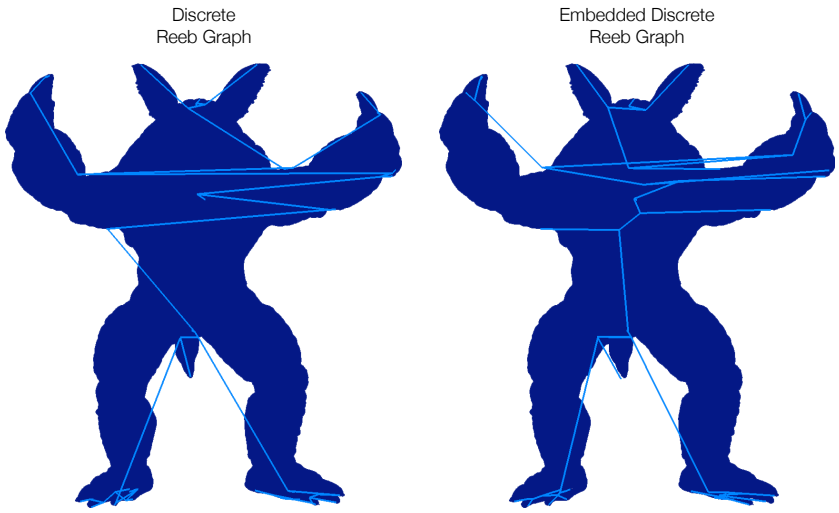


Figure 3.29: *The results of computing the discrete and embedded discrete reeb graph on the Armadillo model. Note how the embedded version better reflects the structure of the model. As every node in the discrete reeb graph must directly correspond to a vertex of the input data set, the nodes cannot move to the center as in the embedded version.*

Figure 3.30 presents a comparison between reeb graphs that were computed with different neighborhood sizes.

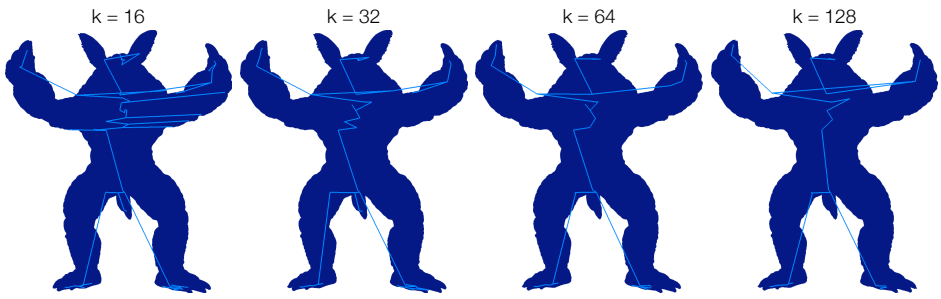


Figure 3.30: *A comparison of reeb graphs created with the weighted 2D algorithm, with a varying size k of neighbors. The more neighbors, the better the algorithm manages to simplify the graph, but processing time rises with increasing size of neighbors.*

Large neighborhood sizes have some negative effects, especially when streaming large data sets, as the active set, memory requirements as well as processing time rise with increasing number of neighbors. However, we can clearly see that the graphs with a higher number of neighbors k have less extraneous nodes.

Embedding Variants

In Figures 3.31, the three embedding variants are shown, with and without filtering. We can see that for the naive embedding, some nodes were simplified away even though they shouldn't have, for example at the head of the armadillo, and a loop from the head of the model to the tips of its ears back into the head has been created. The $2D$ embedding variants do not suffer from this problem.

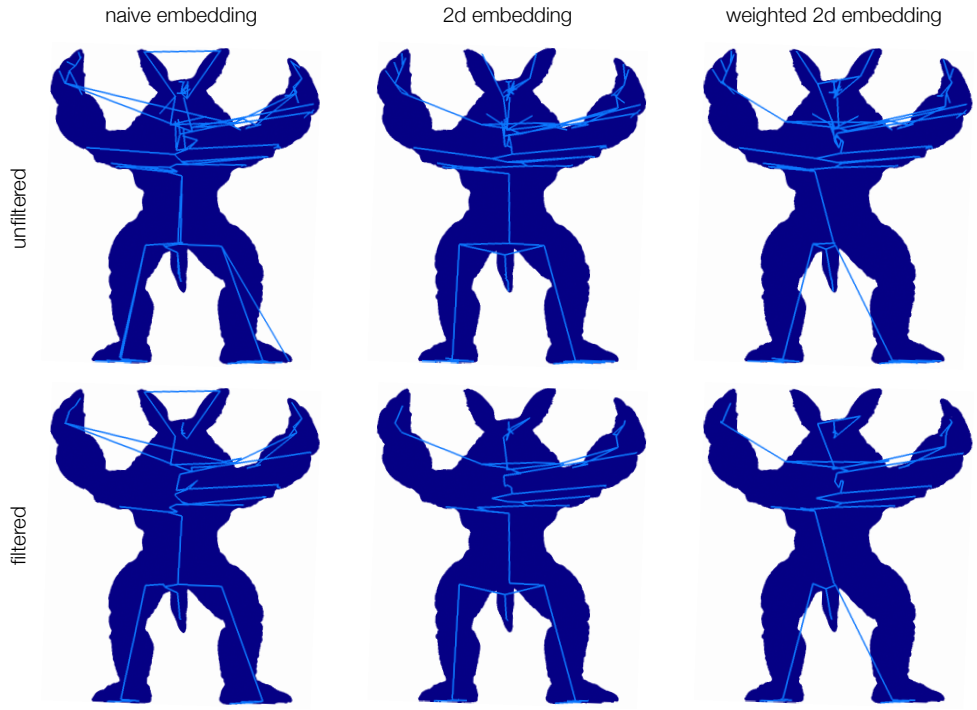


Figure 3.31: *The embedded discrete reeb graphs computed by the reeb operator for the armadillo model.*

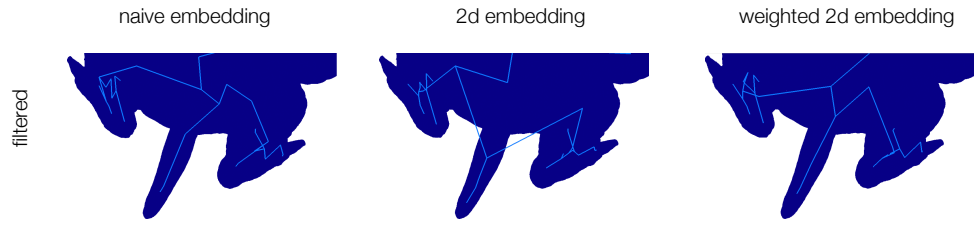


Figure 3.32: *The embedded discrete reeb graphs computed by the reeb operator for the armadillo model. Note how many of the nodes in the weighted $2D$ embedding are centered within the model, while they are closer to the surface for the two other embeddings.*

C H A P T E R 4

STREAMING MESH SIMPLIFICATION

4.1 Overview

The stream processing system is focused on streaming models consisting of point primitives. The major alternative, polygonal triangle or quadrangle meshes, were originally not supported. This chapter presents an extension of the streaming concepts to polygonal meshes and the algorithm that drove this extension, both of which were originally published in [Diaz-Gutierrez et al., 2009]. A final section contains a description of how aspects of this work were integrated back into the stream processing system.

4.2 Background

Polygon mesh sampling is important in many geometry processing problems, including shape approximation, surface reconstruction and parameterization. Correctly sampling surfaces involves choosing a set of points such that their interpolation faithfully reproduces the desired features of the given surface, both in terms of geometry and topology.

The ϵ-net sampling method proposed in [Diaz-Gutierrez et al., 2009] computes a regular sampling of the Gaussian sphere. On contiguous surfaces, the method selects as samples all the points from the input surface whose surface normal coincides with one of those Gaussian sphere samples (a one-to-many mapping from the Gaussian sphere to the given surface).

The sampling method developed by Diaz-Guiterrez and Gopi was used in a collaboration to develop a streaming surface simplification tool. Surface simplification or shape approximation techniques aim at reproducing a given surface with minimum error using fewer mesh elements than in the original. The vertices of a polygonal shape approximation can be considered a sampling of the original surface. Since there is a long history of surface simplification algorithms, we refer to excellent surveys in this field [Heckbert and Garland, 1997; Luebke, 2001]. In general, these methods try to optimize an energy functional or iterate in order to find the (optimal) positions and shape of the mesh elements (vertices, edges and faces) that would reduce the approximation error [Cohen-Steiner et al., 2004; Sheffer, 2001]. The main difference between most of these methods and the one presented here is that they require access to a relatively large amount of connected geometry before deciding when and how to simplify. On the other hand, we decide whether to keep or eliminate a vertex solely based on local information, which makes this method well suited for use in streaming and out-of-core simplification of large meshes, which the rest of this chapter will be focused on. For a more detailed explanation of ϵ-net theory and the necessary proofs please refer to [Diaz-Gutierrez et al., 2009].

4.3 Algorithm

We construct a uniform tessellation of a unit (Gaussian) sphere by iterated regular subdivision of the faces of a regular polyhedron. Let l be the distance between any two adjacent vertices of this tessellation. Given the vertices of a uniform tessellation of a unit (Gaussian) sphere, *our sampling of a surface U is given by the sample set S, which, for smooth manifolds, consists of all those points $x \in U$ whose surface normal $n(x)$ coincides with any of the vertices of the Gaussian sphere tessellation.* Different orientations of the Gaussian sphere produce different sample sets on surface U. However, all these sample sets are equivalent, in the sense that they satisfy the same sampling properties and error bounds.

Direct Mesh Sampling

On a *smooth* manifold M, its samples are all the points on M whose normals coincide with one of the vertices of the Gaussian sphere tessellation G. On a polygonal approximation of M, the samples can be on the mesh faces, edges, or vertices. Since each mesh face represents one normal on the Gaussian sphere, and each mesh edge represents a "curve" of normals, the likelihood of these normals coinciding exactly with a Gaussian vertex is essentially zero. Thus the samples can come only from the input mesh vertex set. A mesh vertex v represents the range of normals N_v that span the interior of a spherical polygon defined by the normals of the mesh faces incident on v. Mesh vertex v is considered a sample if and only if its induced spherical polygon on the Gaussian sphere G contains one or more vertices of G (see Figure 4.1).

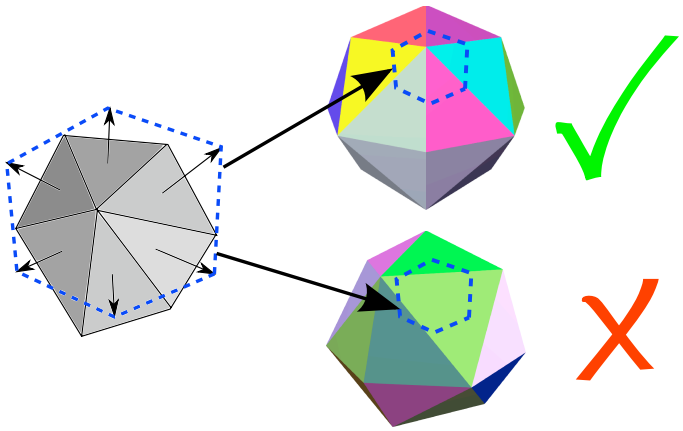


Figure 4.1: *Direct mesh sampling. A mesh vertex is a sample if the spherical polygon formed by its incident face normals contains a Gaussian vertex.*

Unfortunately the spherical polygons created by connecting the normal vectors

of the incident triangles are small (in low curvature regions) and may self-intersect (in saddle vertices). Point location in such spherical polygons is numerically unstable and best avoided. Instead, we proceed as follows.

Conservative Mesh Sampling

A more robust sampling method starts by considering a wide range of mesh vertices as candidate samples, and then discards those that can be positively shown not to map to a sample in the Gaussian sphere tessellation as shown in Figure 4.2.

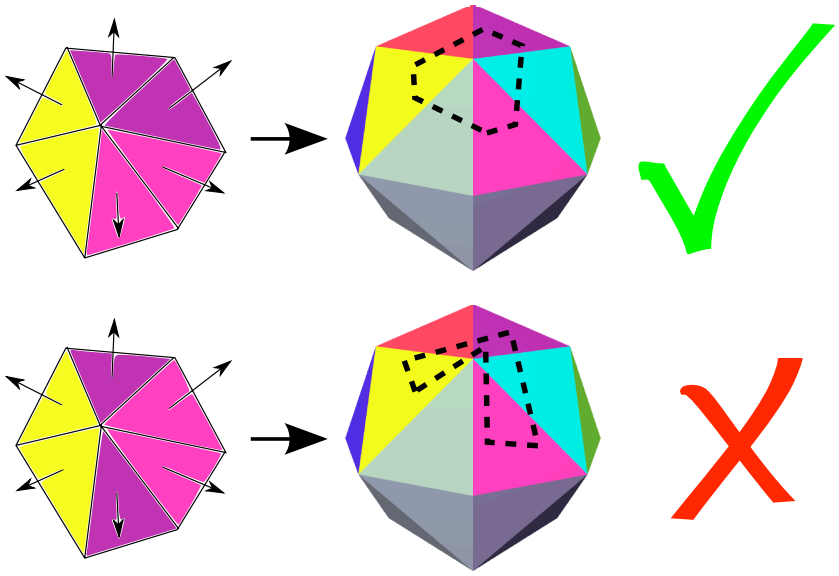


Figure 4.2: *Candidate samples (left) can be pruned if the normals of their incident faces form a spherical polygon (middle) that can be shown not to contain a Gaussian sample just by looking at their associated Gaussian triangles.*

Definition Gaussian triangle association, feature edge and candidate sample: A mesh face t with normal n_t is *associated* with a Gaussian triangle t_G if n_t is located inside t_G in normal space. A mesh edge e is a *feature edge* if its two incident faces are associated with different Gaussian triangles. A mesh vertex v is a *candidate sample* if it is incident on any *feature edge*.

Given a specific tessellation G of the Gaussian sphere, let us assign each mesh face to its associated Gaussian triangle. This would partition the input mesh into regions with the same associated Gaussian triangle. Under this partitioning, it can be seen that the surface samples are a subset of all the mesh vertices on the boundaries between partitions, see Figure 4.3.

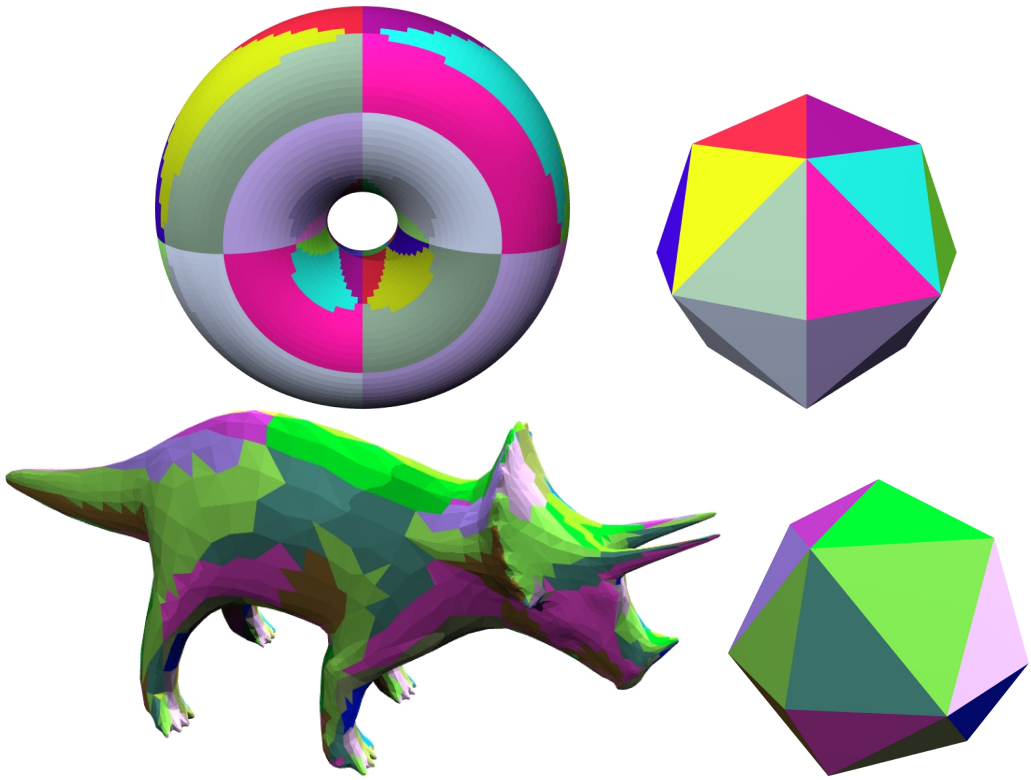


Figure 4.3: *The faces of two meshes (left) are clustered according to their associated Gaussian triangles (Gaussian sphere shown to the right). These clusters are separated by* feature edges. *Vertices adjacent to any* feature edges *are* candidate samples.

We prune the candidate sample set by applying a simple filtering rule as follows.

Identifying non-samples: Not all candidate samples are samples. For example, let a, b, and c be Gaussian triangles such that a is edge-adjacent to b and b is edge-adjacent to c. Let A_1, B_1, C_1, B_2, A_2 be mesh faces incident on a mesh vertex v, in order. Let A_1 and A_2 be associated with a, B_1 and B_2 be associated with b, and C_1 be with c, via their normals. There are feature edges between A_1B_1, B_1C_1, C_1B_2 and B_2A_2, all incident on v. Hence v will be considered a candidate sample. Clearly, the spherical polygon formed by the normal vectors of the mesh faces around v does not enclose any Gaussian vertex and hence v cannot be a sample. Such cases can be generalized as follows (see Figure 4.2). Consider the spherical polygon formed by the normal vectors of the faces incident on the

mesh vertex v. If consecutive vertices of this spherical polygon fall in adjacent [1] Gaussian triangles, and if this spherical polygon can be closed without enclosing a Gaussian sample,then v cannot be a sample. We call these vertices, as well as all the mesh vertices that are inside the partitions, "non-samples". The rest of the mesh vertices are the samples chosen by our algorithm. Figure 4.4 presents a visualization of the samples chosen by the algorithm.

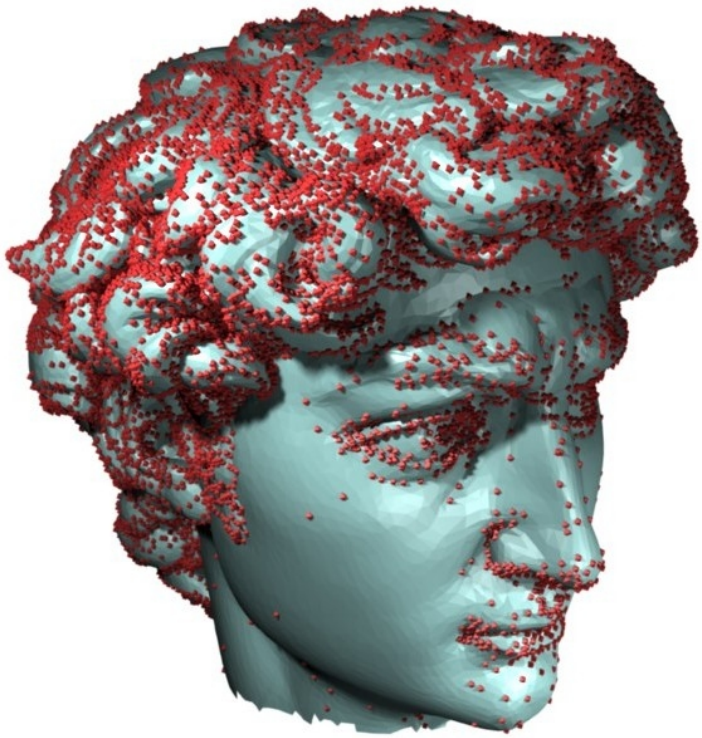


Figure 4.4: *Feature sensitive samples produced by the $\epsilon - net$ sampling algorithm on a large mesh. The Gaussian sphere was tessellated using 112 triangles.*

Removing Non-Samples: All the vertices that are inside the partitions and those vertices on the boundaries of the partitions that are identified as above to be non-samples are removed using simple edge collapse operations [Hoppe, 1996]. All the edges incident on these removable vertices are considered as candidates for edge collapse, and a greedy algorithm on the quadric error of the edges [Garland and Heckbert, 1997] is used to merge these vertices to its neighbor along the collapsed edge. Usual errors associated with edge collapse operations, like flipping of triangle normals and topology changes are checked for, in order to perform a valid edge collapse. The rest of the vertices are considered to be valid samples.

[1]The condition of adjacency through edge connectivity can be relaxed to ensuring that the union of Gaussian triangles traversed by one edge of the spherical polygon form a convex polygon.

4.4 Implementation

trip - triangle processor

The streaming implementation of the above Gaussian sampling and approximation algorithms is based on an underlying *streaming meshes* representation [Isenburg and Lindstrom, 2005] for polygonal surfaces. Similar to the stream-processing approach presented in Chapter 2, the (streaming) triangle mesh is processed sequentially from out-of-core with only a limited amount of data kept active in main memory at any time. As indicated in Figure 4.5, within a sliding window over the streaming mesh data the active vertices and triangles are processed in a multi-stage pipeline of sampling and simplification operators.

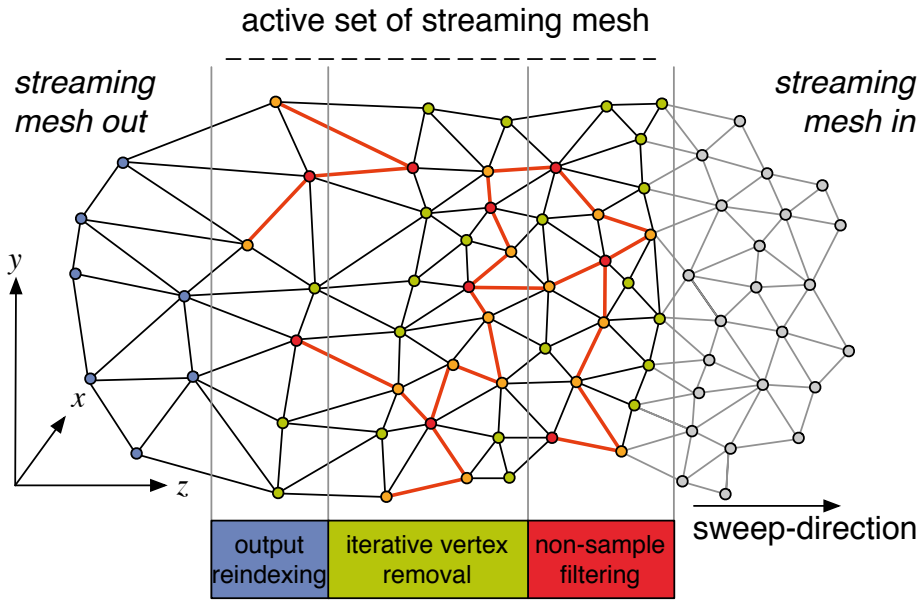


Figure 4.5: *Streaming pipeline with Gaussian sample filtering and non-sample vertex removal.*

The three major phases of the stream-processing pipeline are the identification of sample and non-sample vertices as outlined in the previous section, the removal of non-sample vertices from the input mesh, and the reestablishment of the proper (streaming) mesh output format.

In the first stage, the streaming mesh input is converted into a half-edge triangle mesh data structure [Weiler, 1985] for efficient mesh manipulation in main memory. Care has to be taken to correctly maintain and treat references to future or past mesh elements – in the streaming order – while processing the in-core triangles. Hence the active region of the streaming mesh (see Figure 4.5) is

maintained in-core in a proper topological mesh data structure that dynamically changes as the stream border advances. Moreover, after mesh initialization, each vertex is processed to check if no feature edge is incident on it (inside the partition), or if feature edges are incident on it but the vertex is a non-sample. These operations are local and can thus be applied in a streaming context.

In the second processing stage, non-sample vertices are pruned from the input mesh by the application of mesh simplification operations. Non-sample vertices are removed by an iterative and greedy application of half-edge collapses [Dong et al., 2000]. In order to get a better approximation of the model, we retain a sufficient number of non-samples in-core and repeatedly choose the best half-edge to collapse from all the half-edges that are in-core. Such an approach constitutes an improvement over a greedy method, since each active non-sample vertex gives rise to a number of half-edge collapse candidates. Only half-edge collapses which fulfill a set of mesh topology as well as normal deviation constraints can become collapse candidates. Dynamically maintaining a priority queue of applicable half-edge collapse candidates, prioritized using a quadric error metric [Garland and Heckbert, 1997], the collapse which introduces the smallest error can be selected efficiently.

During the streaming process, triangles in every stage are kept in a priority queue so that faces passed to the next stage are always monotonically ascending according to their minimal vertex corner coordinates along the streaming-axis. When a half-edge is collapsed, its mesh elements (its incident faces and their half-edges) are removed from the mesh immediately. Also, all modified faces (those incident on a removed vertex) are reinserted into the priority queue.

Finally, the last stage is formed by a data cleansing process, in which faces are re-indexed to account for the eliminated non-sample vertices, and non-referenced vertices are omitted from the output.

During the entire process of sampling, edge-collapse, and streaming mesh I/O, the necessary conditions on triangle mesh neighborhood existence are maintained, such as minimum and maximum extents along the streaming-axis for the triangle itself and its one- and two-ring neighborhoods. The output is a simplified mesh, in the same streaming mesh format as the input, where all the vertices are samples. This output can be fed back to the system to iterate this method over a different, random orientation of the Gaussian sphere to ensure the sampling is an ϵ-net on the original mesh.

stream processing system

The stream processing system described in the first part of this thesis was originally only able to handle point streams. After implementing the streaming mesh simplification library, much of the technology was back-ported into the stream

processing system. As shown in Chapter 2, each slice can be processed as long as its reference limits are maintained. Therefore, a secondary face stream be added as long as the face-vertex dependencies are maintained and might potentially increase of the reference limits. The preprocess of the stream processing system was extended to write a second file with face information, and the read and write operators (see Section 2.5.1) were enhanced to process the face information and adjust the reference limits and other flags. A face reindexing step was added to the write operator because of the duplicate vertex removal option.

4.5 Results

Mesh Simplification

A comparison of our results to the QSlim algorithm is presented below. QSlim is an in-core method, and as such has the advantage of having global information on the model. As shown in Table 4.2, when the simplification is extreme, the optimal vertex placement of in-core methods like QSlim cannot be beaten easily. However, even under these unfavorable conditions, the error, measured against the bounding box diagonal, stays reasonably low, as illustrated in Figures 4.6 and 4.7. This result suggests the convenience of a combination of streaming and in-core methods when simplifying huge meshes.

Model	Faces		Error	
	original	simplified	ours	qslim
Armadillo	346K	200K	0.0012	0.00018
Manuscript	4.3M	2.8M	0.00030	0.00026
Dragon	7.2M	3.6M	0.00052	0.00054
Statuette	10M	4.5M	0.0010	0.0015

Table 4.1: *Example comparisons between a simplification satisfying our Gaussian sampling and QSlim. Errors are measured relative to the bounding box diagonal by sampling the Hausdorff distance between the original and the simplified meshes.*

The locality of the sampling method is a great advantage of this system. The effectiveness in that respect can be demonstrated by the memory footprint, or the number of mesh elements, that have to be kept in main memory for sampling and mesh simplification. In Table 4.3 we show the results of the stream-processing implementation described in Section 4.4. As we can see, the memory requirements are very low as the maximum size of mesh elements, of the active set in the sliding window, is only a small fraction of the overall size of the processed models.

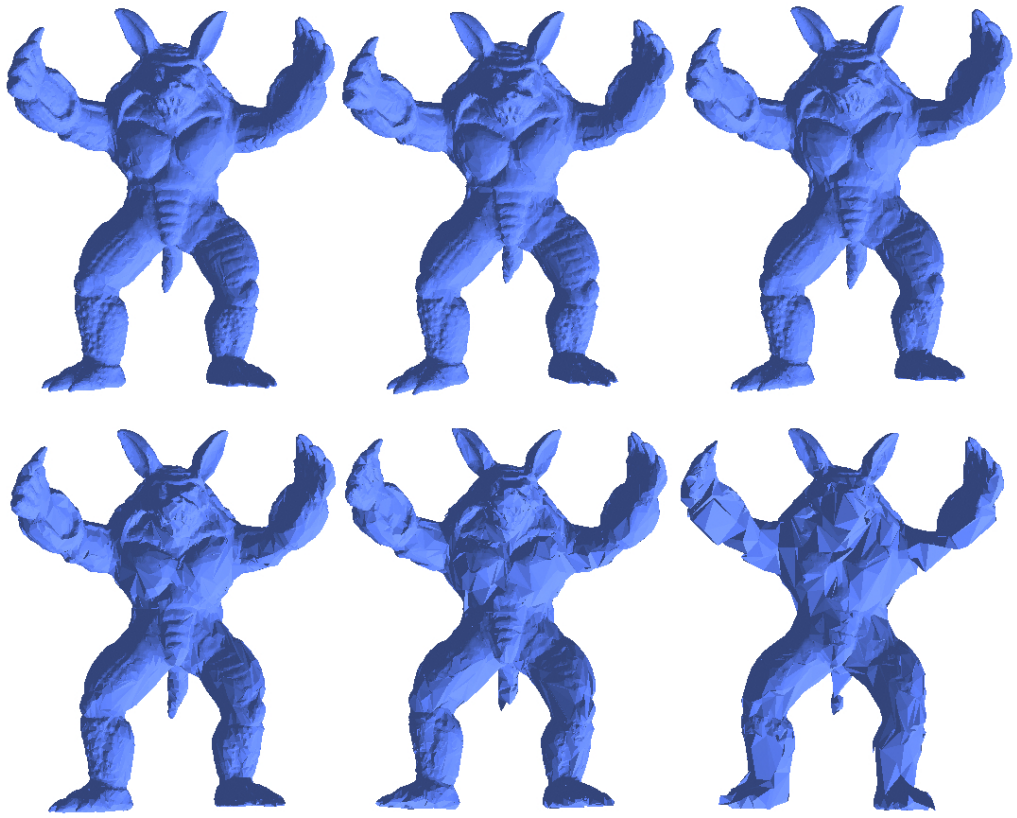


Figure 4.6: *Sequence of simplifications of the armadillo model, with 100%, 34%, 15%, 12%, 10%, and 7% of the original vertices, left to right.*

Integration into the stream processing system

The secondary, parallel face stream might change the behavior of the stream processing system. We found that the main effect of processing an additional face stream is an increase of the active set size. This is to be expected, as triangles may span multiple slices. Table 4.4 shows the active set sizes with and without a secondary face stream. We can see that most models show an increase of the maximum active set size of around 30%. The difference between the models can be explained by two factors: the extent of the largest triangle in streaming direction and the active set required to compute the neighbors. Models with a large increase either contain large triangles or have an advantageous structure for neighborhood computation.

#faces	Approximation error	
	Our method	QSlim
52703	0.0035	0.0013
43924	0.0042	0.0014
35146	0.0049	0.0014
26369	0.0065	0.0017

Table 4.2: *Comparison of the approximation error on the Armadillo model (345944 faces originally) using our method and QSlim. Each row is coarser than the previous, as indicated by the number of vertices. Errors are measured relative to the bounding box diagonal by sampling the Hausdorff distance between the original and the simplified meshes.*

Model	Max. Active Set Size	
	2 subdivisions	4 subdivisions
Armadillo	43016	49480
Manuscript	91893	63702
Dragon	394324	467161
Statuette	418581	495466

Table 4.3: *Maximal size of the active set during stream-processing various meshes, given for a Gaussian sampling with sphere subdivision 2 and 4 respectively.*

Summary

To summarize, we present an novel system for the streaming simplification of polygonal meshes based on the ϵ-nets and the Gaussian sphere. The system is capable of processing large meshes efficiently, and the quality of the simplified meshes compares favorably to established in-core methods such as QSlim.

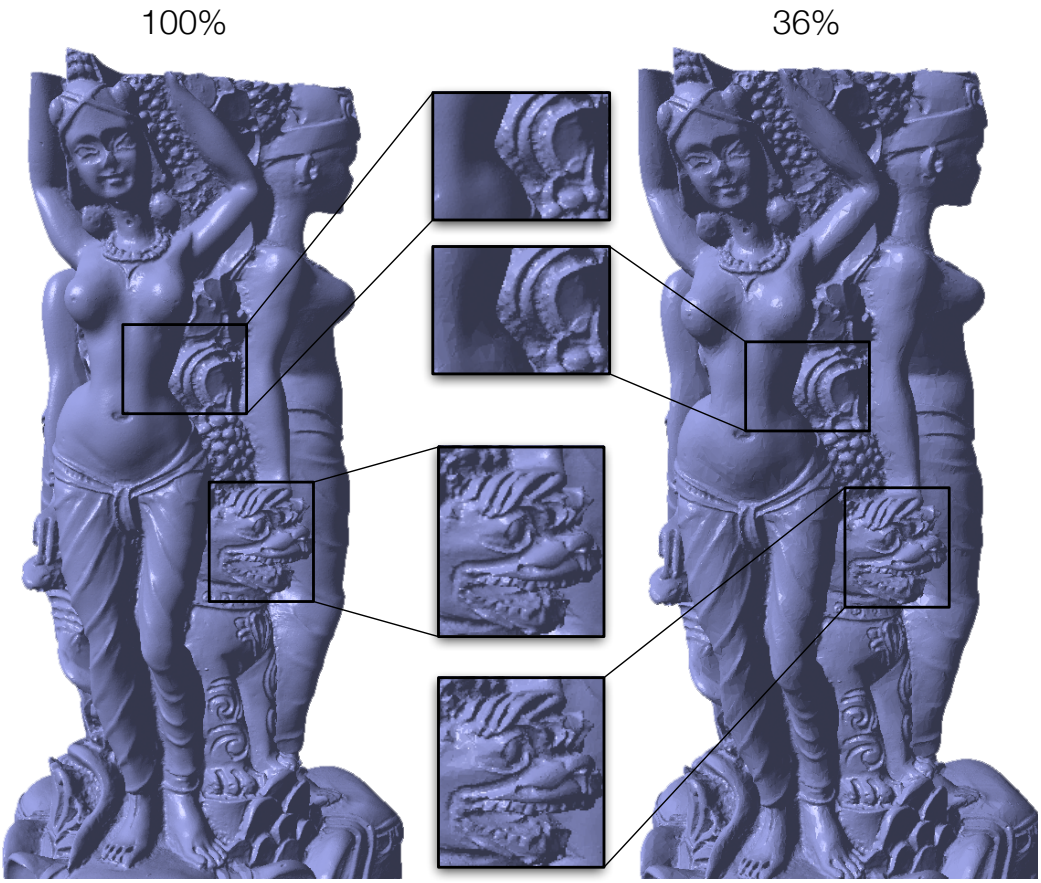


Figure 4.7: *Mesh simplification on the Statuette model. The model on the right is reduced to 36% of the data, but only few artifacts are visible. Both images were rendered with flat shading to highlight differences.*

Model	Vertices	Faces	Max. AS Points	Max. AS Faces	Increase
Bunny	35'947	69'451	4'810	7'960	39.0%
Armadillo	172'982	345'944	7940	14'210	44.12%
Happy Buddha	543'652	1'087'716	29'728	42'516	30.08%
Dragon	3'609'600	7'219'045	362'590	413'860	13.3%
Statuette	4999'996	10'000'000	117'370	165'670	29.15 %

Table 4.4: *Comparison between the maximum active set sizes with and without streaming face information in the* stream processing system *using a neighbor-only processing chain. Smaller increases are better, as the active set size is indicative of memory usage.*

C H A P T E R 5

CONCLUSIONS

5.1 Summary

In this thesis, a system to process large geometric point and triangle mesh data sets is presented. New methods to compute geometric properties such as Gaussian curvature are discussed and combined with various existing algorithms to form a complete geometry processing solution, including a novel method to compute and embed reeb graphs for point sets. We introduce a technique for efficient and type-safe access to run-time-specified arbitrary point attributes in C++, an operator concept that allows transparent multi-threading and removes dependencies between the various operators, and demonstrate the efficiency of the system on large geometric data sets.

The developed stream processing system encapsulates various algorithms for computing or estimating geometric properties in stream operators or chain operators. The processing pipeline can be specified at run-time, and the various operators are linked together in a fashion conceptually similar to unix pipes. Each operator requires a set of input attributes, e.g. a point surface normal, and writes out the new attributes it computed. The only dependencies between the various stream operators are therefore their respective inputs and outputs, so that e.g. normal orientation can be performed if any previous operator writes a normal attribute. The run-time structures concept that enables this functionality allows a programmer to type-safely and efficiently access arbitrary point attributes from within an operator, and the processing chain can be run in single-precision, mixed-precision or double-precision modes transparently. Operators that encapsulate algorithms which do not maintain active-set-global data structures such as spatial trees can be transparently multithreaded, and an algorithm developer does not have to take are of any thread or data access synchronization.

Various algorithms that compute or estimate geometric properties are discussed, implemented, and examined with respect to quality and performance. Algorithms for the estimation of the point neighborhood, elliptical surface splats that can be used for point-based rendering and fitted spheres are integrated into the system. Novel or improved algorithms are introduced for the estimation of normal direction and orientation, point radius and accurate Gaussian curvature. The accuracy of our Gaussian curvature estimation method is shown on synthetic models where the theoretical Gaussian curvature is known. We also show visualizations of the Gaussian curvature visualized on scanned models. The method and results of a discrete reeb graph computation on point data sets are presented, and multiple reeb graph embedding methods are examined.

The $\epsilon - net$ sampling method is described and exploited to develop a novel method for the streaming simplification of large polygonal meshes, and the results of our simplification technique are discussed with respect to visual quality and run-time performance. The polygonal mesh streaming was implemented based on

our stream processing system and later reintegrated to enable the stream processing system to process on multiple streams in parallel and provide operators for both points and polygonal meshes. The results on the impact this extension has on the behavior of the stream processing system are shown and discussed.

5.2 Directions for Future Work

The results and contributions presented in this thesis form an auspicious foundation for further research. Ideas for improvements and promising directions for future work are discussed below.

- *GPU Acceleration*

  The only part of the stream processing system that is currently GPU-accelerated is the visualization. Today's GPUs and powerful GPU programming languages such as OpenCL[1] and CUDA[2] promise to greatly improve processing speed.

  The encapsulation inherent in the current design with the stream operators would benefit from porting demanding algorithms to the GPU. Rendering kernels could be used independently from within operators, and the dependent data is readily and easily available already.

  Especially neighborhood detection, currently the most costly operator, could greatly benefit from a GPU implementation. As even the multi-threading is currently bound by the running time of the most demanding non-multi-threadable operator, optimizing the neighborhood operator offers the largest improvement.

- *Normal Orientation*

  Normal orientation would be improved by using a multi-pass approach, where a data set is streamed multiple time through an orientation operator. A global graph of oriented patches in the vain of the current reeb graph implementation could be maintained and used to make normal patches consistent. The current normal orientation operator can process a model such as the statuette with 10 million points into 20 patches, so the corresponding graph would be small enough to keep in memory even for huge data sets.

[1] [Khronos, 2010]
[2] [nVidia, 2010]

· *Streaming Axis*

Instead of using one of the coordinate system axis as streaming axis, the system could be extended to use Morton order, a space-filling curve. This would reduce likely reduce the size of the active set and therefore improve run-time performance and memory requirements.

· *Reeb Graph*

The reeb graph operator could be extended with an additional constraints system that enforces that no edge of the reeb graph can pass through the surface. The current algorithm has the property that with enough simplification of the reeb graph, the edges might leave the actual model and connect e.g. a reeb vertex in the hand of a humanoid model directly to one in the torso, without going up the arm.

· *Mesh Simplification*

The results achieved with the stream mesh simplification are excellent, but the system lacks a fine-grained but intuitive mechanism for the user to control simplification. Using the sphere subdivision as control parameter is suboptimal since in order to reach a high compression/simplification ratio, a Gaussian sphere with subdivision 2 or 1 is practically required. Additional parameters that can be tweaked are non-intuitive.

An improvement for the preprocess would be the cutting of triangles that span a large segment on the stream axis. This reduces memory usage and the size of the active set. Without this step, a worst-case model with a triangle that contains both the vertex with the smallest and with the largest component in streaming direction can break the system, as the active set would equal the total data set.

A limitation of the current system is its dependence on manifold meshes. Algorithms for removal of non-manifold subsets or streaming correction of a defective the mesh would greatly enhance the applicability on real-world data sets.

· *Points and Polygonal Meshes*

With the integration of multiple-stream capability into the stream processing system as a result of the work on the trip streaming mesh simplification system, there is a chance for benefitting from the intersection of point-based and mesh-based methods. Streaming meshing operators would allow generating manifold meshes from scanned point data sets that could in turn be simplified using the Gaussian sphere approach.

· *Additional Operators*

The efficiency and low programming overhead of developing new stream operators should be used to develop and implement additional algorithms that work on point data sets. The independence of the single operators means that operators offering improvements over existing algorithms as well as operators solving new problems both benefit the system. One example is correct normal orientation, which in turn would benefit sphere fitting and curvature estimation, another example is hole detection in combination with splat estimation that might lead to visual improvements in rendering with smaller cost due to less splat overlap.

The concepts and techniques used in the stream processing system have proven to be a powerful tool for the processing of huge data sets, with the largest model tested requiring only 0.5% or less of it's total size in memory at any time for a full processing chain. Curvature estimation provides excellent principal curvatures and directions, we can compute embedded reeb graphs of point-only models and our streaming mesh simplification efficiently reduces model size of large models while keeping the shape close to the original. We believe that the work presented in this thesis will benefit future research, including but not limited to point processing, geometry processing and the stream-processing of large data sets.

BIBLIOGRAPHY

[YBS, 2004] (2004). A fast and simple stretch-minimizing mesh parameterization. In *Shape Modeling Int.*, pages 200–208, Washington, DC, USA. IEEE Computer Society.

[boo, 2007] (2007). Boost c++ libraries. Website.

[Adamson and Alexa, 2006] Adamson, A. and Alexa, M. (2006). Anisotropic point set surfaces. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, AFRIGRAPH '06, pages 7–13, New York, NY, USA. ACM.

[Agam and Tang, 2005] Agam, G. and Tang, X. (2005). A sampling framework for accurate curvature estimation in discrete surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):573–583.

[Alexa and Adamson, 2009] Alexa, M. and Adamson, A. (2009). Interpolatory point set surfaces–convexity and hermite data. *ACM Trans. Graph.*, 28:20:1–20:10.

[Alexa et al., 2001] Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. (2001). Point set surfaces. In *IEEE Vis.*, pages 21–28, Washington, DC, USA. IEEE Computer Society.

[Alexa et al., 2003] Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. (2003). Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15.

[Allegre et al., 2007] Allegre, R., Chaine, R., and Akkouche, S. (2007). A streaming algorithm for surface reconstruction. In *Proceedings Eurographics Symposium on Geometry Processing*, pages 79–88.

[Amenta et al., 1998] Amenta, N., Bern, M., and Kamvysselis, M. (1998). A new Voronoi-based surface reconstruction algorithm. In *SIGGRAPH*, pages 415–421.

[Amenta and Kil, 2004] Amenta, N. and Kil, Y. J. (2004). Defining point-set surfaces. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 264–270, New York, NY, USA. ACM.

[Anwar, 2009] Anwar, S. (2009). Reeb graph for point sets. Technical report, Self-published, http://sites.google.com/site/sabatanwar/discretereebgraph.

[Attene et al., 2001] Attene, M., Biasotti, S., and Spagnuolo, M. (2001). Re-meshing techniques for topological analysis. In *Proceedings of the International Conference on Shape Modeling & Applications*, pages 142–, Washington, DC, USA. IEEE Computer Society.

[Bajaj et al., 1997] Bajaj, C., Pascucci, V., and Schikore, D. (1997). The contour spectrum. In *Visualization '97., Proceedings*, pages 167 –173.

[Bentley, 1975] Bentley, J. (1975). Multidimensional binary search trees used for associative searching.

[Bernardini et al., 1999] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *IEEE Vis.*, 5(4).

[Biasotti et al., 2000] Biasotti, S., Mortara, M., and Spagnuolo, M. (2000). Surface compression and reconstruction using reeb graphs and shape analysis. In *spring conference on computer graphics*.

[Biasotti and Ricerche, 2001] Biasotti, S. and Ricerche, C. N. D. (2001). Topological techniques for shape understanding. In *In Central European Seminar on Computer Graphics, CESCG*, page 01.

[Bolitho et al., 2007] Bolitho, M., Kazhdan, M., Burns, R., and Hoppe, H. (2007). Multilevel streaming for out-of-core surface reconstruction. In *Proceedings Eurographics Symposium on Geometry Processing*, pages 69–78.

[Bösch and Pajarola, 2008] Bösch, J. and Pajarola, R. (2008). A system for stream processing of point data. Technical Report IFI-2008.03, Department of Informatics, University of Zürich.

[Bösch and Pajarola, 2009] Bösch, J. and Pajarola, R. (2009). Flexible configurable stream processing of point data. In *International Conference on Computer Graphics, Visualization and Computer Vision (WSCG)*. UNION Agency, Science Press.

[Botsch et al., 2005] Botsch, M., Hornung, A., Zwicker, M., and Kobbelt, L. (2005). High-quality surface splatting on today's gpus. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 0:17–141.

[Botsch and Kobbelt, 2003] Botsch, M. and Kobbelt, L. (2003). High-quality point-based rendering on modern GPUs. In *Proceedings Pacific Graphics 2003*, pages 335–343. IEEE, Computer Society Press.

[Botsch et al., 2002] Botsch, M., Wiratanaya, A., and Kobbelt, L. (2002). Efficient high quality rendering of point sampled geometry. In *Proceedings Eurographics Workshop on Rendering*, pages 53–64.

[Carr et al., 2000] Carr, H., Snoeyink, J., and Axen, U. (2000). Computing contour trees in all dimensions. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 918–926, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Carr et al., 2004] Carr, H., Snoeyink, J., and van de Panne, M. (2004). Simplifying flexible isosurfaces using local geometric measures. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 497–504, Washington, DC, USA. IEEE Computer Society.

[Cheng et al., 2004] Cheng, L., Bhushan, A., Pajarola, R., and El Zarki, M. (2004). Real-time 3D graphics streaming using MPEG-4. In *Proceedings IEEE/ACM Workshop on Broadband Wireless Services and Applications*.

[Cohen-Steiner et al., 2004] Cohen-Steiner, D., Alliez, P., and Desbrun, M. (2004). Variational shape approximation. *ACM ToG.*, 23(3):905–914.

[Cole-McLaughlin et al., 2004] Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., and Pascucci, V. (2004). Loops in reeb graphs of 2-manifolds. *Discrete Comput. Geom.*, 32:231–244.

[Cuccuru et al., 2009] Cuccuru, G., Gobbetti, E., Marton, F., Pajarola, R., and Pintus, R. (2009). Fast low-memory streaming MLS reconstruction of point-sampled surfaces. In *Proceedings Graphics Interface*, volume 27, pages 15–22.

[Dachsbacher et al., 2003] Dachsbacher, C., Vogelgsang, C., and Stamminger, M. (2003). Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662.

[Dai et al., 2007] Dai, M., Newman, T. S., and Cao, C. (2007). Least-squares-based fitting of paraboloids. *Pattern Recogn.*, 40:504–515.

[de Berg et al., 1997] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1997). *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin.

[Denning, 1970] Denning, P. J. (1970). Virtual memory. *ACM Computing Surveys*, 2(3):153–189.

[Dey, 2006] Dey, T. K. (2006). *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis (Cambridge Monographs on Applied and Computational Mathematics)*. Cambridge University Press, New York, NY, USA.

[Dey and Goswami, 2004] Dey, T. K. and Goswami, S. (2004). Provable surface reconstruction from noisy samples. In *Sympos. Comput. Geom.*, pages 330–339. ACM.

[Diaz-Gutierrez et al., 2009] Diaz-Gutierrez, P., Bösch, J., Pajarola, R., and Gopi, M. (2009). Streaming surface sampling using gaussian ϵ-nets. *The Visual Computer*, 25(5-7):411–421. The original publication is available at www.springerlink.com.

[Dong et al., 2000] Dong, W., Li, J., and Kuo, J. (2000). Fast mesh simplification for progressive transmission. In *Proceedings International Conference on Multimedia and Expo ICME 2000*. IEEE.

[Engel et al., 2000] Engel, K., Sommer, O., and Ertl, T. (2000). A framework for interactive hardware-accelerated remote 3D-visualization. In *Proceedings EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, pages 167–177.

[Floater, 1997] Floater, M. S. (1997). Parametrization and smooth approximation of surface triangulations. *Comput. Aided Geom. Des.*, 14(3):231–250.

[Funkhouser et al., 2005] Funkhouser, T., Kazhdan, M., Min, P., and Shilane, P. (2005). Shape-based retrieval and analysis of 3d models. *Commun. ACM*, 48:58–64.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

[Garland and Heckbert, 1997] Garland, M. and Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *Proceedings ACM SIGGRAPH*, pages 209–216. ACM SIGGRAPH.

[Gobbetti and Marton, 2004] Gobbetti, E. and Marton, F. (2004). Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(1):815–826.

[Gopi et al., 2000] Gopi, M., Krishnan, S., and Silva, C. (2000). Surface Reconstruction using Lower Dimensional Localized Delaunay Triangulation. *Eurographics*, 19(3):467–478.

[Gross and Pfister, 2007a] Gross, M. and Pfister, H., editors (2007a). *Point-Based Graphics*. Morgan Kaufmann Publishers - Elsevier.

[Gross, 2006] Gross, M. H. (2006). Getting to the point...? *IEEE Computer Graphics and Applications*, 26(5):96–99.

[Gross and Pfister, 2007b] Gross, M. H. and Pfister, H., editors (2007b). *Point-Based Graphics*. Morgan Kaufmann Publishers - Elsevier.

[Grossman and Dally, 1998] Grossman, J. and Dally, W. J. (1998). Point sample rendering. In *Proceedings Eurographics Rendering Workshop 98*, pages 181–192. Eurographics.

[Guennebaud and Gross, 2007] Guennebaud, G. and Gross, M. (2007). Algebraic point set surfaces. *ACM Trans. Graph.*, 26.

[Guskov and Wood, 2001] Guskov, I. and Wood, Z. J. (2001). Topological noise removal. In *No description on Graphics interface 2001*, GRIN'01, pages 19–26, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

[Heckbert and Garland, 1997] Heckbert, P. S. and Garland, M. (1997). Survey of polygonal surface simplification algorithms. Technical report, Computer Science Department, Carnegie Mellon University.

[Henney, 2000] Henney, K. (2000). Valued conversions. *C++ Report*, 12(7):37–40.

[Hétroy and Attali, 2003] Hétroy, F. and Attali, D. (2003). Topological quadrangulations of closed triangulated surfaces using the reeb graph. *Graph. Models*, 65:131–148.

[Hilaga et al., 2001] Hilaga, M., Shinagawa, Y., Kohmura, T., and Kunii, T. L. (2001). Topology matching for fully automatic similarity estimation of 3d shapes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 203–212, New York, NY, USA. ACM.

[Hoppe, 1996] Hoppe, H. (1996). Progressive meshes. In *SIGGRAPH*, pages 99–108. ACM SIGGRAPH.

[Hoppe et al., 1992] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. (1992). Surface reconstruction from unorganized points. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 71–78, New York, NY, USA. ACM.

[Hübner et al., 2006] Hübner, T., Zhang, Y., and Pajarola, R. (2006). Multi-view point splatting. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 285–294, New York, NY, USA. ACM.

[Isenburg and Lindstrom, 2005] Isenburg, M. and Lindstrom, P. (2005). Streaming meshes. In *Proceedings IEEE Visualization*, pages 231–238.

[Isenburg et al., 2003] Isenburg, M., Lindstrom, P., Gumhold, S., and Snoeyink, J. (2003). Large mesh simplification using processing sequences. In *Proceedings IEEE Visualization*, pages 465–472. Computer Society Press.

[Isenburg et al., 2005] Isenburg, M., Lindstrom, P., and Snoeyink, J. (2005). Streaming compression of triangle meshes. In *Proceedings Eurographics Symposium on Geometry Processing*, pages 111–118.

[Isenburg et al., 2006] Isenburg, M., Liu, Y., Shewchuk, J., and Snoeyink, J. (2006). Streaming computation of delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056.

[Jones et al., 2004] Jones, T. R., Durand, F., and Zwicker, M. (2004). Normal improvement for point rendering. *IEEE Computer Graphics and Applications*, 24(4):53–56.

[Khodakovsky et al., 2003] Khodakovsky, A., Litke, N., and Schröder, P. (2003). Globally smooth parameterizations with low distortion. In *SIGGRAPH*, pages 350–357, New York, NY, USA. ACM Press.

[Khronos, 2010] Khronos (2010). Opencl.

[Knuth, 1998] Knuth, D. E. (1998). *The Art of Computer Programming, 3rd Edition*. Addison-Wesley.

[Kobbelt and Botsch, 2004] Kobbelt, L. and Botsch, M. (2004). A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814.

[LaBRI, 2011] LaBRI, L. B. d. R. e. I. (2011). http://tulip.labri.fr/tulipdrupal/.

[Levoy and Whitted, 1985] Levoy, M. and Whitted, T. (1985). The use of points as display primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill.

[Linderman, 1996] Linderman, J. P. (1996). rsort and fixcut. man pages. revised June 2000.

[Liu et al., 2002] Liu, G. H., Wong, Y. S., Zhang, Y. F., and Loh, H. T. (2002). Adaptive fairing of digitized point data with discrete curvature. *Computer Aided Design*, 32(4):309–320.

[Luebke, 2001] Luebke, D. P. (2001). A developer's survey of polygonal simplification algorithms. *IEEE Comput. Graph. Appl.*, 21(3):24–35.

[Meenakshisundaram, 2001] Meenakshisundaram, G. (2001). *Theory and Practice of Sampling and Reconstruction for Manifolds with Boundaries*. PhD thesis, Department of Computer Science, University of North Carolina Chapel Hill.

[Miao et al., 2007] Miao, Y.-W., Feng, J.-Q., Xiao, C.-X., Peng, Q.-S., and Forrest, A. (2007). Differentials-based segmentation and parameterization for point-sampled surfaces. *Journal of Computer Science and Technology*, 22:749–760. 10.1007/s11390-007-9088-5.

[Mitra and Nguyen, 2003] Mitra, N. J. and Nguyen, A. (2003). Estimating surface normals in noisy point cloud data. In *Proceedings Symposium on Computational Geometry*, pages 322–328. ACM.

[Mitra et al., 2004] Mitra, N. J., Nguyen, A., and Guibas, L. (2004). Estimating surface normals in noisy point cloud data. In *special issue of International Journal of Computational Geometry and Applications*, volume 14, pages 261–276.

[Noimark and Cohen-Or, 2003] Noimark, Y. and Cohen-Or, D. (2003). Streaming scenes to MPEG-4 video-enabled devices. *IEEE Computer Graphics and Applications*, 23(1):58–64.

[nVidia, 2010] nVidia (2010). Cuda.

[Pajarola, 2003] Pajarola, R. (2003). Efficient level-of-details for point based rendering. In *Proceedings IASTED Invernational Conference on Computer Graphics and Imaging (CGIM)*.

[Pajarola, 2005] Pajarola, R. (2005). Stream-processing points. In *Proceedings IEEE Visualization*, pages 239–246.

[Pajarola et al., 2004] Pajarola, R., Sainz, M., and Guidotti, P. (2004). Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics*, 10(5):598–608.

[Pajarola et al., 2005] Pajarola, R., Sainz, M., and Lario, R. (2005). XSplat: External memory multiresolution point visualization. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP)*, pages 628–633.

[Pascucci et al., 2007] Pascucci, V., Scorzelli, G., Bremer, P.-T., and Mascarenhas, A. (2007). Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26.

[Pauly and Gross, 2001] Pauly, M. and Gross, M. (2001). Spectral processing of point-sampled geometry. In *Proceedings ACM SIGGRAPH*, pages 379–386. ACM Press.

[Pauly et al., 2002a] Pauly, M., Gross, M., and Kobbelt, L. P. (2002a). Efficient simplification of point-sampled surfaces. In *IEEE Vis*, Washington, DC, USA. IEEE Computer Society.

[Pauly et al., 2002b] Pauly, M., Gross, M., and Kobbelt, L. P. (2002b). Efficient simplification of point-sampled surfaces. In *Proceedings IEEE Visualization*, pages 163–170. Computer Society Press.

[Pauly et al., 2003] Pauly, M., Keiser, R., Kobbelt, L., and Gross, M. (2003). Shape modeling with point-sampled geometry. *ACM Transactions on Graphics*, 22(3):641–650.

[Pfister and Gross, 2004] Pfister, H. and Gross, M. (2004). Point-based computer graphics. *IEEE Computer Graphics and Applications*, 24(4):22–23.

[Pfister et al., 2000] Pfister, H., Zwicker, M., van Baar, J., and Gross, M. (2000). Surfels: Surface elements as rendering primitives. In *Proceedings ACM SIGGRAPH*, pages 335–342. ACM SIGGRAPH.

[Pratt, 1987] Pratt, V. (1987). Direct least-squares fitting of algebraic surfaces. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 145–152, New York, NY, USA. ACM.

[Praun et al., 2001] Praun, E., Sweldens, W., and Schröder, P. (2001). Consistent mesh parameterizations. In *SIGGRAPH*, pages 179–184, New York, NY, USA. ACM Press.

[Reeb, 1946] Reeb, G. (1946). Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus Acad. Sciences*, 222:847–849.

[Ren et al., 2002] Ren, L., Pfister, H., and Zwicker, M. (2002). Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings EUROGRAPHICS*, pages 461–470. also in Computer Graphics Forum 21(3).

[Rusinkiewicz, 2004] Rusinkiewicz, S. (2004). Estimating curvatures and their derivatives on triangle meshes. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 486–493, Washington, DC, USA. IEEE Computer Society.

[Rusinkiewicz and Levoy, 2000] Rusinkiewicz, S. and Levoy, M. (2000). QSplat: A multiresolution point rendering system for large meshes. In *Proceedings ACM SIGGRAPH*, pages 343–352. ACM SIGGRAPH.

[Rusinkiewicz and Levoy, 2001] Rusinkiewicz, S. and Levoy, M. (2001). Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proceedings Symposium on Interactive 3D Graphics*, pages 63–68. ACM SIGGRAPH.

[Sainz and Pajarola, 2004] Sainz, M. and Pajarola, R. (2004). Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879.

[Sainz et al., 2004] Sainz, M., Pajarola, R., and Lario, R. (2004). Points reloaded: Point-based rendering revisited. In *Proceedings Symposium on Point-Based Graphics*, pages 121–128. Eurographics/IEEE VGTC.

[Samet, 2006] Samet, H. (2006). *Foundations of Multidimensiona and Metric Data Structures*. Morgan Kaufmann Publishers - Elsevier.

[Sander et al., 2003] Sander, P. V., Wood, Z. J., Gortler, S. J., Snyder, J., and Hoppe, H. (2003). Multi-chart geometry images. In *SGP*, pages 146–155, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Sheffer, 2001] Sheffer, A. (2001). Model simplification for meshing using face clustering. *CAD*, 33:925–934.

[Shinagawa and Kunii, 1991] Shinagawa, Y. and Kunii, T. L. (1991). Constructing a reeb graph automatically from cross sections. *IEEE Comput. Graph. Appl.*, 11:44–51.

[Shinagawa et al., 1991] Shinagawa, Y., Kunii, T. L., and Kergosien, Y. L. (1991). Surface coding based on morse theory. *IEEE Comput. Graph. Appl.*, 11:66–78.

[Shinagawa et al., 1995] Shinagawa, Y., Kunii, T. L., Sato, H., and Ibusuki, M. (1995). Modeling contact of two complex objects, with an application to characterizing dental articulations. *Computers Graphics*, 19(1):21 – 28. Visual Computing.

[Sorkine et al., 2002] Sorkine, O., Cohen-Or, D., Goldenthal, R., and Lischinski, D. (2002). Bounded-distortion piecewise mesh parameterization. In *IEEE VIS*, pages 355–362, Washington, DC, USA. IEEE Computer Society.

[Steiner and Fischer, 2001] Steiner, D. and Fischer, A. (2001). Topology recognition of 3d closed freeform objects based on topological graphs. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, SMA '01, pages 305–306, New York, NY, USA. ACM.

[Takahashi et al., 1997] Takahashi, S., Shinagawa, Y., and Kunii, T. L. (1997). A feature-based approach for smooth surfaces. In *Proceedings of the fourth ACM symposium on Solid modeling and applications*, SMA '97, pages 97–110, New York, NY, USA. ACM.

[Van Loon, 2007] Van Loon, J. (2007). Implementation of a web interface for a stream-based point processing application. Bachelor Thesis.

[Vitter, 2001] Vitter, J. S. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271.

[Vo et al., 2007] Vo, H. T., Callahan, S. P., Lindstrom, P., Pascucci, V., and Silva, C. T. (2007). Streaming simplification of tetrahedral meshes. *IEEE Transaction on Visualization and Computer Graphics*, 13(1):145–155.

[Weber et al., 2007] Weber, G. H., Dillard, S. E., Carr, H., Pascucci, V., and Hamann, B. (2007). Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13:330–341.

[Weiler, 1985] Weiler, K. (1985). Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40.

[Wendland, 1995] Wendland, H. (1995). Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv. Comput. Math.*, 4(1):389–396.

[Weyrich et al., 2004] Weyrich, T., Pauly, M., Keiser, R., Heinzle, S., Scandella, S., and Gross, M. (2004). Post-processing of scanned 3D surface data. In *Proceedings Symposium on Point-Based Graphics*, pages 85–94. Eurographics/IEEE VGTC.

[Wood et al., 2004] Wood, Z., Hoppe, H., Desbrun, M., and Schröder, P. (2004). Removing excess topology from isosurfaces. *ACM Trans. Graph.*, 23:190–208.

[Wu and Kobbelt, 2003] Wu, J. and Kobbelt, L. (2003). A stream algorithm for the decimation of massive meshes. In *Proceedings Graphics Interface*, pages 185–192.

[Zhang et al., 2005] Zhang, E., Mischaikow, K., and Turk, G. (2005). Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24:1–27.

[Zhang and Pajarola, 2006a] Zhang, Y. and Pajarola, R. (2006a). GPU-accelerated transparent point-based rendering. In *ACM SIGGRAPH Sketches*, page 178.

[Zhang and Pajarola, 2006b] Zhang, Y. and Pajarola, R. (2006b). Single-pass point rendering and transparent shading. In *Proceedings Symposium on Point-Based Graphics*, pages 37–48. Eurographics/IEEE VGTC.

[Zhang and Pajarola, 2007] Zhang, Y. and Pajarola, R. (2007). Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics*, 31(2):175—189.

[Zwicker et al., 2004] Zwicker, M., Räsänen, J., Botsch, M., Dachsbacher, C., and Pauly, M. (2004). Perspective accurate splatting. In *Proceedings of Graphics Interface 2004*, GI '04, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

C H A P T E R A

APPENDIX

Test Machine Specification

Figure A.1: *Technical details about the machine that was used for all performance tests in this thesis.*